

## Word-Python Project

### Rev. 1

The goal of this project is to create a Python3 script, utilizing the python-docx package, that generates one or more Word documents from an input Word file and an input Excel file whose filepaths (directory path and filename) are hard-coded into the Python script. Example Word and Excel files are provided with this description. The Word file is an ordinary Word document that contains “paragraphs” and “tables” (these terms have a detailed meaning within the internal format of a Word document so see a more detailed discussion of paragraphs and tables below) that further include “tags” whose format and function are defined in this document.

The Python script parses each paragraph and table within the Word file, using classes and methods provided by the python-docx package, to find tags that will be replaced with content defined by each tag. A “TEXT” tag is replaced with specific text defined in the Excel spreadsheet. An “IMAGE” tag is replaced with an image whose filepath, image height and image width are defined in the Excel spreadsheet. A “FILE” tag is replaced with the content of the Word file whose filepath is defined in the Excel spreadsheet.

Note that a FILE tag may identify a Word file with nested tags; i.e., TEXT, IMAGE and additional FILE tags may exist within the Word file identified by the FILE tag. Nesting up to a hard-coded nesting limit (e.g., 10 levels) is allowed or we can choose not to enforce a nesting limit (subject only to memory limits) if that would be easier to implement.

Please note these helpful resources:

- <https://stackoverflow.com/questions/24805671/how-to-use-python-docx-to-replace-text-in-a-word-document-and-save>
- <https://github.com/python-openxml/python-docx/issues/519>
- <https://github.com/python-openxml/python-docx/issues/156>
- <https://python-docx.readthedocs.io/en/latest/user/quickstart.html#adding-a-table>
- <https://python-docx.readthedocs.io/en/latest/user/quickstart.html#adding-a-picture>
- [https://python-docx.readthedocs.io/en/latest/api/table.html#docx.table.\\_Cell.add\\_paragraph](https://python-docx.readthedocs.io/en/latest/api/table.html#docx.table._Cell.add_paragraph)
- <https://stackoverflow.com/questions/48713465/python-docx-copy-table>

### Parsing and Modifying a Word Document (Paragraphs, Runs and Tables)

This section of the project description describes how the python-docx package handles Word document parsing and modification. Understanding this section is critical to understanding the project.

In the discussion that follows, words starting with capital letters reflect data structures and the same words starting with lower-case letters reflect the word in its ordinary use. For example, “Paragraph” reflects a data structure with that name while “paragraph” reflects the usual meaning of that term, within the context of Microsoft Word, such as a series of formatted characters with a carriage return.

## Paragraphs and Runs

Documentation for the python-docx package describes that Word documents contain “Paragraphs” and “Tables” (two data structure types). A Paragraph contains one or more “Run” data structures. A Run is a collection of characters with consistent formatting.

Understanding how Runs relate to Paragraphs is critical to understanding how Word documents internally store their data and for completing this project. The python-docx package provides logic for parsing the internal structure of a Word document, such as retrieving Runs within Paragraphs and retrieving Paragraphs or Tables within a Word document.

An example of Runs within a Paragraph should explain the significance of a Run. If a paragraph contains the formatted text “Hello *this* is a **paragraph**”, with the word “this” in italics and the word “paragraph” in bold, the Paragraph (data structure) storing the paragraph (collection of formatted characters) contains a first Run for “Hello ” (note the space character at the end), a Run for “*this*” (formatted in italics), a Run for “ is a ” (note the space character at the start and end of these characters), and a Run for “**paragraph**”.

As a programming exercise, if we wanted to replace all instances of the word “paragraph” with the word “sentence” within a Word document, we would programmatically retrieve a list of Paragraphs from the document, we would iterate through the list of Paragraphs, for the current (iterating) Paragraph we would retrieve a list of Runs from the current Paragraph, we would iterate through the list of Runs for the current Paragraph, and we would search and replace the text of the current Run.

When searching for text within a Word document, one complexity could be that the text crosses a Run boundary. For example, when searching for the word “paragraph” in the example above, the document could contain the following text: “Hello *this* is a **paragraph**” (note how “para” is in bold but “graph” is not in bold. In this example paragraph, the “para” characters are stored in a first Run and the “graph” characters are stored in a second Run.

This project avoids this complexity by requiring that all tags have consistent formatting through the tag, so that a tag does not cross a Run boundary. For example, the `[[TEXT:text1]]` tag discussed below will be parsed so that “`[[TEXT:text1]]`” (consistent formatting) is recognized as a tag but that “`[[TEXT:text1]]`” (inconsistent formatting within the tag) is not parsed as a tag. The requirement for consistent formatting within a tag simplifies this project because tags will not cross Run boundaries.

The python-docx modules provide functions for creating a Paragraph, creating a Run, retrieving Runs from a Paragraph, adding (appending) a Run to (the end of) a Paragraph, and adding (appending) a Paragraph to (the end of) a document. Adding a Run to a Paragraph appends the Run to a list of Runs within the Paragraph (we can only append a Run to the end of the list). Adding a Paragraph to a Word document also appends the Paragraph to a list of Paragraphs within the Word Document (we can only append a Paragraph to the end of the list).

The python-docx package does not provide an easy way to replace a Run, within a Paragraph, within a Word document, with another Run. Instead, a developer must write a function to accomplish the replacement goal using the append capability discussed above.

These suggested steps will replace one instance of “target-text”, in the Word document, with “replacement-text” (you can do things differently if you have a better way to accomplish text replacement):

- Big Step 1 - find a Run containing content as “target-text” to replace within a Word document by iterating through each Run within each Paragraph in the Word document. The Run containing “replacement-text” is “the Run to replace” in the steps that follow.
- Big Step 2 – prepare data for text replacement
  - Small Step 2-1: Create a first list of Paragraphs prior to the Paragraph containing the Run to replace
  - Small Step 2-2: Create a second list of Paragraphs after the Paragraph containing the Run to replace
  - Small Step 2-3: Create a first list of Runs in the Paragraph prior to the Run to replace
  - Small Step 2-4: Create a second list of Runs in the Paragraph after the Run to replace
  - Small Step 2-5: Create a new Run, from the Run to be replaced, and change the text within the new Run as needed to accomplish the desired text replacement
- Big Step 3 – create a list of Paragraphs reflecting the text replacement
  - Small Step 3-1: Create an empty list of paragraphs that will store the document’s content after text replacement (called the new list of Paragraphs)
  - Small Step 3-2: Add the first list of Paragraphs (above) to the new list of Paragraphs
  - Small Step 3-3: Create a new Paragraph that will contain the replaced text. Add the first list of Runs to the new Paragraph. Add the new Run to the new Paragraph. Add the second list of Runs to the new Paragraph
  - Small Step 3-4: Add the new Paragraph to the new list of Paragraphs
  - Small Step 3-5: Add the second list of Paragraphs to the new list of Paragraphs

After the steps above, the new list of Paragraphs represents the modified content in the Word document. The prior list of Paragraphs can be discarded.

The steps above would be repeated for all instances of “text” within a Word document.

When implementing this project, please create a function that will perform the steps above for a text string, provided as an argument, and for a list of Paragraphs provided as a second argument (you would probably create this function without the suggestion).

### **Tables and Paragraphs**

Data for a table is stored in a Table (data structure). A Table contains a list of Rows, with each Row containing data for cells within the row of a table. The content for each cell of a Table is a Paragraph (as discussed above). Therefore, a Table has rows and cells containing Paragraphs.

The python-docx package provides a function that returns a list of Tables within a Word document. That package also has functions for adding a Table to a Paragraph.

Replacing text within a table in a Word document is conceptually similar to replacing text within a list of paragraphs. More specifically, text replacement in a table involves iterating through the rows in a table, iterating through the cells in each row, finding text matching the text to be replaced (in a Run within a Paragraph within a cell) and replacing that text. A replacement table will be formed from the rows prior to the matching Run, from a modified version of the row and cells containing the matching Run, and from rows after the matching Run.

A Table is added to a Word document by adding (appending) the Table to a Paragraph (just as for adding a Run to a Paragraph). Thus, the process for replacing text in a Table involves creating a new version of the Table with the modified text, creating a new list of Paragraphs before the Paragraph containing the Table, adding the Paragraphs before the Paragraph containing the table to the new list of Paragraphs, creating a replacement Paragraph (including the modified Table) for the Paragraph that contained the Table and adding the replacement Paragraph to the new list of Paragraphs, creating a new list of Paragraphs after the Paragraph containing the Table, and adding the list of Paragraphs after the Paragraph containing the Table to the new list of Paragraphs to form a new version of the Word document with replaced table text.

When implementing this project, please create a function that will perform the steps above for a text string, provided as an argument, and for Table provided as a second argument. This function will be called by another function that iterates through all Tables in the Word document.

The conceptual similarities between replacing text in a table (cell) and replacing text within a Paragraph are pretty straightforward (although Tables are a little more involved due to rows and cells containing Paragraphs).

For simplicity, this project will assume that a Table's cell does not contain a nested table. Although Word allows nested tables, we will ignore that possibility for parsing purposes.

### **Images and Paragraphs**

An image is stored within a Run within a Paragraph. Replacing an IMAGE tag (text) with its corresponding image is very similar to the text replacement examples discussed above except that the modified Run has an image appended at the appropriate location within the modified Run (rather than text or a Table).

The python-docx package provides a function for adding an image to a Run. Therefore, a replacement Run is created by creating a Run instance, adding the text before the IMAGE tag to the replacement Run, adding the image for the IMAGE tag to the replacement Run, and adding the text after the IMAGE tag to the replacement Run. The replacement Run is added to the Word document as discussed above for text replacement.

### **Tag Format Generally**

All tags use the same general format. All tags start with two opening brackets (“[[“), end with two closing brackets (“]]”), must include a tag type identifier (“TEXT”, “IMAGE” or “FILE”), must include a tag value identifier, and optionally include a comment set off by parenthesis (“(this is a comment)”). Examples of each tag type are shown below.

- [[TEXT:text1(this is a comment)]]
- [[IMAGE:fooimage3(this is also a comment)]]
- [[FILE:foo\_file(this is additionally a comment)]]

In the examples above, “text1”, “fooimage3” and “foo\_file” are tag value identifiers for the TEXT, IMAGE and FILE tags, respectively, and values for those tags are present in the provided spreadsheet as discussed separately below.

## Tag Replacement Generally

An example of tag replacement is shown next. In this example, the provided Word file contains the following content:

This is an example Word file.

A TEXT tag is shown here and its content is [[TEXT:text1]].

An IMAGE tag is shown here and its content is [[IMAGE:image1]].

A FILE tag is shown here and its content is “[[FILE:file1]].”

For this example, let’s say that the corresponding spreadsheet includes the value of “text1” as “Test Text”, the spreadsheet includes a filepath for “image1” containing the image shown below, and the spreadsheet includes a value of “file1” with content shown further below.

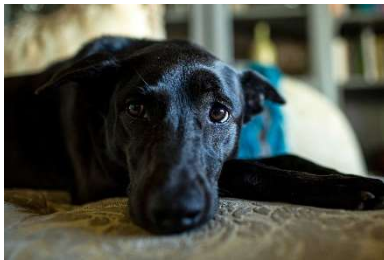


Image file content:


Word file content:

**This is an example nested Word file**

The Word file generated by the Python script, for the current example, is seen below.

This is an example Word file.

A TEXT tag is shown here and its content is Test Text.



An IMAGE tag is shown here and its content is

A FILE tag is shown here and its content is **"This is an example nested Word file."**

As seen above, the TEXT, IMAGE and FILE tags have been replaced with their corresponding content. Note how content around the tags, as well as spacing and formatting of the content around the tags, has been preserved.

### IF-ELSE-ENDIF (Conditional) Logic

Content can also be conditionally inserted within the Word file by enclosing that content within an IF-ELSE-ENDIF (conditional) tag expression. The condition tested by the IF tag is a named "GLOBAL" value read from the spreadsheet as discussed below. If the value read from the spreadsheet for the conditional tag is True, the IF-ELSE-ENDIF tags are replaced with content between the IF and ELSE tags. If the value read from the spreadsheet for the conditional tag is False, the IF-ELSE-ENDIF tags are replaced with content between the ELSE and ENDIF tags. Use of an IF tag always requires corresponding ELSE and ENDIF tags; i.e., the ELSE tag is not optional (although its content could be empty).

An example of Word file with conditional logic is shown below.

This is an example of a Word file with conditional logic.

Today is [[IF:condition1]]a good day[[ELSE]]a bad day[[ENDIF]] because  
[[IF:condition1]] today is [[TEXT:day\_of\_the\_week]][[ELSE]]the month is  
[[TEXT:current\_month]][[ENDIF]]

For this example, assume that day\_of\_the\_week is Tuesday and current\_month is November.

If condition1 is true, the resulting Word file is shown below.

This is an example of a Word file with conditional logic.

Today is a good day because today is Tuesday

If condition1 is false, the resulting Word file is shown below.

This is an example of a Word file with conditional logic.

Today is a bad day because the month is November

IF-ELSE-ENDIF tags can include all tag types including nested file tags and nested IF-ELSE-ENDIF tags. For example, nesting a second IF tag within a first IF tag allows for operating on two conditions as shown below.

This is an example of a Word file with nested conditional logic.

The status of conditions 1 and 2 are: [[IF:condition1]][[IF:condition2]]Both True[[ELSE]]True and False[[ENDIF]][[ELSE]] [[IF:condition2]]False and True[[ELSE]]Both False[[ENDIF]][[ENDIF]]!

In the example above, the replaced text is unique for the particular combination of condition1 and condition2 values. Nesting of IF-ELSE-ENDIF tags should be possible up to a hard-coded nesting limit (e.g., 10) or of unlimited nesting (subject to memory limits) if that would be easier to implement.

Therefore, IF-ELSE-ENDIF tags allow flexible conditional content insertion.

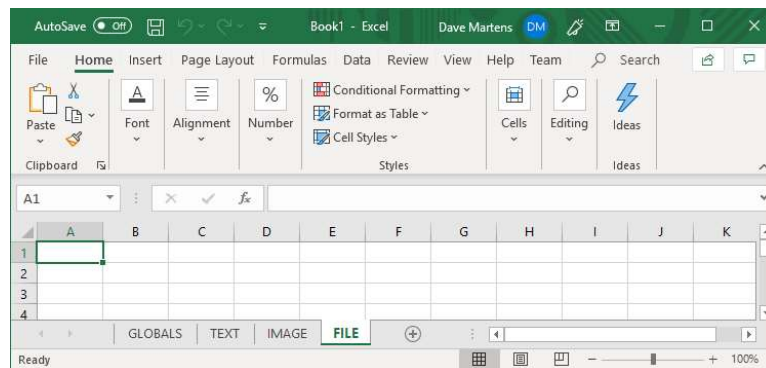
### **Storing the Generated Word File**

The Word file generated by the Python script is stored at a filepath (directory path and file name) defined for the GLOBAL value “DESTINATION”. As discussed below, DESTINATION has a filepath value for each of the one or more Word files generated by running the Python script.

### **Spreadsheet Provides Tag Values**

As discussed at the start of this document, the Python script reads a Word file and an Excel spreadsheet file whose filepaths are hard-coded into the script. The previous section of this document discussed the Word file and this section of the document discusses the spreadsheet.

The purpose of the spreadsheet is to store one or more values for each of the TEXT, IMAGE, FILE and GLOBAL identifiers within Word document. Values for each of these identifier types are included within separate “tabs” (worksheets) within the Excel file as seen below.



The spreadsheet will be a conventional XLSX spreadsheet rather than a CSV file so a simple capability for reading XLSX spreadsheets is needed; see, e.g., the popular “xlrd” package [at https://xlrd.readthedocs.io/en/latest/](https://xlrd.readthedocs.io/en/latest/).

Let’s keep spreadsheet reading and validating as simple as possible. We can perform a few simple checks to ensure that major problems do not exist with the spreadsheet and can make some simplifying assumptions to spreadsheet-reading aspects of the Python script manageable.

The spreadsheet will be validated to ensure that all four tabs/sheets shown above exist. The order of these sheets within an Excel workbook is not important.

### **GLOBALS Sheet**

The GLOBALS sheet includes columns for each global identifier (including at least DESTINATION and ITERATOR) and rows for each set of values. The ITERATOR identifier will be used to reference value sets within the TEXT, IMAGE and FILE sheets as discussed and illustrated in greater detail below. The GLOBALS sheet also includes columns for each conditional tag.

An example GLOBALS sheet is shown below for a Word file that includes conditionals CONDITION1 and CONDITION2.



	A	B	C	D	E	F	G	H	I
1									
2	ITERATOR	DESTINATION	CONDITION1	CONDITION2					
3	1	c:/Dir1/file1.docx	TRUE	TRUE					
4	2	c:/Dir1/file2.docx	TRUE	FALSE					
5	3	c:/Dir2/file1.docx	FALSE	TRUE					
6	4	c:/Dir2/file3.docx	FALSE	FALSE					
7									
8									
9									
10									

As seen above, a header row identifying the columns exists and columns named ITERATOR, DESTINATION, CONDITION1 and CONDITION2 exist.

The content within the ITERATOR column starts immediately below its header and can be any value that is unique within that column. An error is raised if no iterator values exist below the column header or if an iterator value is duplicated within the iterator values.

The content within the DESTINATION column starts immediately below its header, continues for the duration of the iterator rows, and can be any string that is unique within that column (legal filepaths will not be checked in this version of the script). An error is raised if the number of destination filepaths does not match the number of iterator values, or if a filepath value is duplicated within the filepath values.

The content within additional columns for conditional tags, if any, starts immediately below the column header for those columns, continues for the duration of the iterator rows and must be true or false (only). Header names for the additional columns must match identifier names for `[[IF:identifier]]` tags within the Word document. An error is raised if a one-to-one correspondence does not exist between identifiers for IF tags and column headers for global identifiers; i.e., column headers for conditionals must be unique and must exactly match the number and name of corresponding IF tags in the Word document.

Parsing of the GLOBALS sheet begins by finding the one occurrence of the ITERATOR column heading within the GLOBALS sheet. To keep spreadsheet parsing simple, we require that the ITERATOR column is the left-most column among other populated spreadsheet cells (but does not need to be the left-most column in the worksheet), the DESTINATION column is immediately to the right of the ITERATOR column, and all identifiers for IF tags are immediately follow to the right of the ITERATOR column. Content outside of the columns specified by the column headers described above and the rows set by the ITERATOR column is ignored.

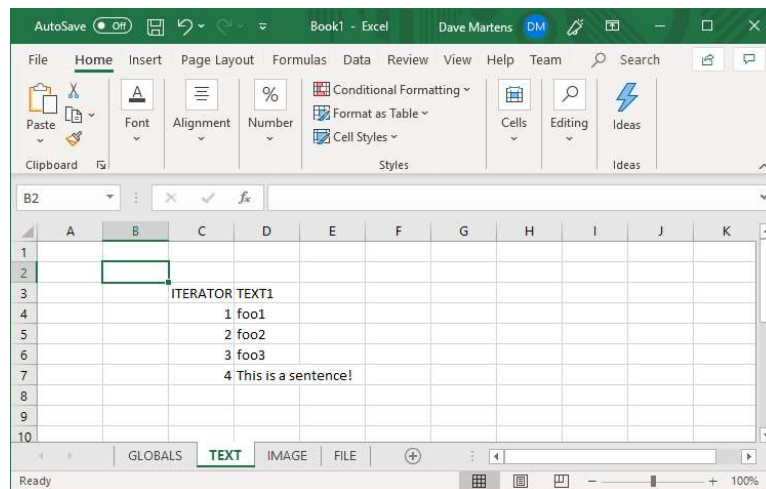
No error or warning is reported if a conditional tag column exists within the GLOBALS sheet, but a corresponding IF tag does not exist within the Word document because the corresponding IF tag could exist within a FILE that was not incorporated due to other conditional logic. However, an

error is raised if an IF tag exists in the Word document and no corresponding column, containing a column header for the value within the IF tag, exists within the GLOBALS worksheet.

In summary, parsing of the GLOBALS sheet starts by finding the ITERATOR column heading (the only cell whose content is “ITERATOR”), determining how many column heading cells exist, determining how many cells exist under the ITERATOR column heading cell, and parsing the grid of cells defined by the determined rows and columns.

## TEXT Sheet

Parsing of the other sheets is similar. An example of a TEXT sheet for the four iterator values discussed above is shown below.



As seen above, the same ITERATOR column exists (with the same iterator values as seen in the GLOBALS sheet), and a TEXT1 column exists with values for all iterator values. This example assumes that only a single TEXT tag (whose identifier is “TEXT1”) exists; additional columns would exist if other TEXT tags were present within the Word document.

No error or warning is reported a column for a TEXT tag exists within the TEXT worksheet, but a corresponding TEXT tag does not exist within the Word document because the TEXT tag could exist within a FILE that was not incorporated due to other conditional logic. However, an error is raised if a TEXT tag exists in the Word document and no corresponding column exists within the TEXT worksheet.

## IMAGE Sheet

An example of an IMAGE sheet is shown below.

The screenshot shows an Excel spreadsheet with the following data:

	ITERATOR	IMAGE1_filepath	IMAGE1_width	IMAGE1_height
1	1	C:/DATA/image1.png	2	4
2	2	C:/DATA/image3.png	2.5	-1
3	3	C:/DATA/image5.png	-1	4
4	4	C:/DATA/image7.png	-1	-1

As seen above, the same ITERATOR column exists (with the same iterator values as seen in the GLOBALS sheet), and three columns exist to support a single IMAGE tag (with “IMAGE1” identifier) with values for all iterator values. The three columns are named “<identifier>\_filepath”, “<identifier>\_width” and “<identifier>\_height”. E.g., the width and height values are dimensions in inches; e.g., width and height for image1.png is 2” wide and 4” high.

Width or height values of “-1” reflect default or scaled dimensions. If width and height are both “-1”, Word will place the image in the Word document at its natural width and height. If width is “-1” and height is some positive value, Word will place the image in the Word document at the specified height and will scale the image’s width proportionally. Similarly, if height is “-1” and width is some positive value, Word will place the image in the Word document at the specified width and will scale the image’s height proportionally. Thus, an image can be set to a specific height and width, a scaled height and width, or the image’s natural height and width through height and width dimensions including “-1”.

This example assumes that only a single IMAGE tag (whose identifier is “IMAGE1”) exists; additional sets of three columns would exist if other IMAGE tags were present within the Word document.

No error or warning is reported if a set of three columns, for a single image, exists within the IMAGE worksheet but a corresponding IMAGE tag does not exist within the Word document because the IMAGE tag could exist within a FILE that was not incorporated due to other conditional logic. However, an error is raised if an IMAGE tag exists in the Word document and no corresponding column exists within the IMAGE worksheet.

## FILE Sheet

An example of a FILE sheet is shown below.

	A	B	C	D	E	F	G	H
1								
2								
3		ITERATOR	FILE1	FILE2				
4			1 C:/DATA/document1.docx	C:/DATA/document2.docx				
5			2 C:/DATA/document3.docx	C:/DATA/document4.docx				
6			3 C:/DATA/document5.docx	C:/DATA/document6.docx				
7			4 C:/DATA/document7.docx	C:/DATA/document8.docx				
8								
9								
10								

As seen above, the same ITERATOR column exists (with the same iterator values as seen in the GLOBALS sheet), and a column exists to support each of two FILE tags (with “FILE1” and “FILE2” identifiers) with values (filepaths) for all iterator values.

No error or warning is reported if a column for a FILE exists within the FILE worksheet, but a corresponding FILE tag does not exist within the Word document because the FILE tag could exist within a FILE that was not incorporated due to other conditional logic. However, an error is raised if a FILE tag exists in the Word document and no corresponding column exists within the FILE worksheet.

## Word Document Parsing

Documentation for the python-docx module seems quite good and stackoverflow.com includes many useful examples for reading and writing Word files with that module. Therefore, the complexity of this project seems moderate – neither hard nor easy.

As discussed in module documentation (<https://python-docx.readthedocs.io/en/latest/#api-documentation>), an existing Word document can be read, for further analysis, with the first line of the following code. The second line obviously stores the Word document as a new file.

```
document = Document('existing-document-file.docx')
document.save('new-file-name.docx')
```

As seen above, an instance of the Document class is returned from parsing the file. One part of parsing an existing Word file is retrieving the Paragraph and Table instances from the file through the “paragraph” and “table” attributes, each of which contain a list of corresponding class instances. These instances reflect the top-level content instances within a Document instance.

A Paragraph instance contains text and/or images. Each Paragraph includes one or more Run instances, each of which seems to reflect a series of formatted text or images. See <https://python-docx.readthedocs.io/en/latest/api/text.html#docx.text.paragraph.Paragraph> and <https://python-docx.readthedocs.io/en/latest/api/text.html#docx.text.run.Run>. For example, the string “Hello

there you” within a Paragraph in a Document seems to comprise a Run for “Hello ”, a Run for “there”, a Run for “ ” (a single space) and a Run for “you”.

A simplifying assumption that we will make in this project is that format changes will not occur within tags so that a tag will never span two Run instances. Therefore, tag parsing involves searching and replacing tags within each Run instance within each Paragraph instance (and Table instance as discussed next) within the Document.

Each Table instance comprises one or more rows with one or more cells (columns). Each cell includes at least a Paragraph instance comprising one or more Run instances as discussed above or a (nested) Table instance.

A very relevant StackOverflow discussion on searching-and-replacing nested content with python-docx is in the 2<sup>nd</sup> response here: <https://stackoverflow.com/questions/24805671/how-to-use-python-docx-to-replace-text-in-a-word-document-and-save>.

### **Error Conditions to Check**

The following error conditions will be checked and reported (error message and program exist):

- Spreadsheet or Word file hard-coded in Python file does not exist
- Spreadsheet lacks any of these tabs: GLOBALS, TEXT, IMAGE or FILE
- IF tag is present but ELSE or ENDIF tag not found
- Improperly formatted TEXT, IMAGE, FILE, or IF-ELSE-ENDIF tags including comments
- GLOBALS tab lacks DESTINATION tag for storing resulting Word file
- GLOBALS tab lacks ITERATOR tag for distinguishing iterations in TEXT, IMAGE and FILE tabs
- GLOBAL identifier exists in IF tag but not present in GLOBALS tab (or value is neither true nor false)

### **Summary**

The principles discussed above should describe the goals of this project and a starting point for achieving those goals with the python-docx module. A successful Python script for this project will implement the features described herein without changes unless changes are discussed in advance. I’m very open to better ways to achieving the stated goals but any improvements must not compromise the script’s results in other ways.