**Knesset Speaker: Identifying speaker names in the Knesset corpus and matching them to actual Knesset members**

Alon Sokolovski  
[alon.sokolovski@gmail.com](mailto:alon.sokolovski@gmail.com)

Contents

[**1. Intro** 1](#_Toc175440752)

[**2. Parsing the documents** 2](#_Toc175440753)

[**2.1. Reason** 2](#_Toc175440754)

[**2.1.1. Ensuring Accurate Text Parsing** 2](#_Toc175440755)

[**2.1.2. Leveraging Document Metadata for Speaker Identification** 3](#_Toc175440756)

[**2.1.3. Granular Control and Error Handling** 3](#_Toc175440757)

[**2.1.4. Enhanced Speaker Identification through Contextual Understanding** 3](#_Toc175440758)

[**2.2. Implementation** 3](#_Toc175440759)

[**2.2.0. Name Validation** 4](#_Toc175440760)

[**2.2.1. Load Document** 5](#_Toc175440761)

[**3. Metadata** 35](#_Toc175440762)

[**4. Matching speakers to actual Knesset members** 35](#_Toc175440763)

[**5. Splitting to sentences** 35](#_Toc175440764)

[**6. Output Format** 35](#_Toc175440765)

[**7. Testing** 35](#_Toc175440766)

# **1. Intro**

The Knesset Corpus is a massive dataset that captures the proceedings of the Knesset, Israel's parliament, including both plenary and committee deliberations from the past 30 years. In this project, I developed methods for accurately identifying and correcting speaker names across these deliberations. Given the various ways in which names can be written, the challenge was to ensure that every mention of a Knesset member, despite differences in name presentation, is consistently linked to their texts. Additionally, it was crucial to prevent other fragments of text from being mistakenly identified as speaker names.

I had access to a table containing information about all Knesset members (approximately 1,000 individuals) and a processed version of the Knesset Corpus, where sentences were already linked to their speakers. However, there were some inaccuracies in this initial identification process. My task was to navigate through these records, identify, and correct errors in speaker identification. For example, instances where "בני גנץ" and "בנימין גנץ" were mistakenly treated as separate entities needed to be consolidated, ensuring that all contributions were accurately attributed to the correct Knesset member. This effort was crucial in creating a more reliable and useful resource for anyone interested in the workings of the Knesset.

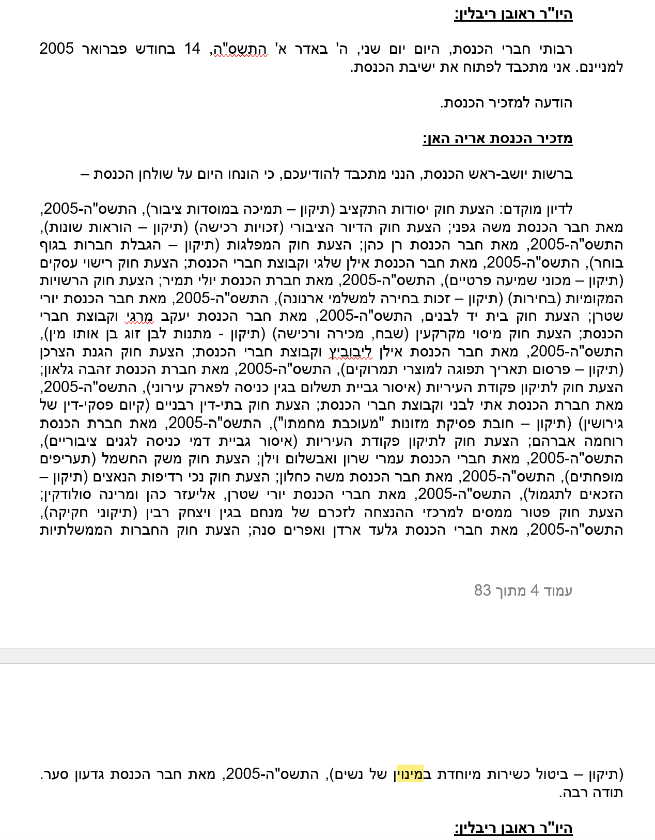
# **2. Parsing the documents**

## **2.1. Reason**

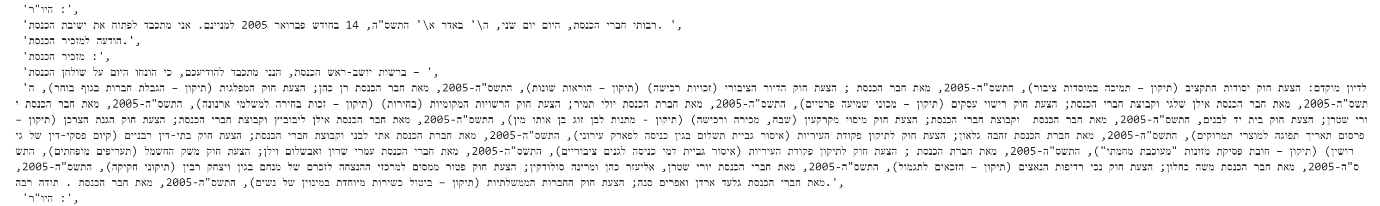
Despite having access to a pre-processed version of the Knesset Corpus, I decided to parse the documents on my own for several critical reasons:

### **2.1.1. Ensuring Accurate Text Parsing**

The commonly used **python-docx** library, which is the standard tool for parsing .docx files, was not parsing files correctly in some instances. I noticed that it sometimes missed words, leading to inaccuracies in the processed data. This issue was significant enough to warrant implementing my own parsing solution for both Windows and Linux platforms using MS Word or LibreOffice respectively, which ensured that the entire content was accurately captured, including all words, formatting, and metadata, providing a more reliable foundation for subsequent analysis.

****

***Screenshot 1:*** *‘16\_ptm\_129044.docx’ opened in MS Word (Windows)*

***Screenshot 2:*** *16\_ptm\_129044.docx as it was parsed by python-docx, missing speaker names*

By implementing a custom parser, I was able to accurately process the documents without the risk of missing or misinterpreting text, providing a more reliable foundation for subsequent analysis.

### **2.1.2. Leveraging Document Metadata for Speaker Identification**

In addition to the textual content, the documents contained valuable metadata, such as underlines and text alignment (centered, left, or right), which I used to accurately identify speakers. This metadata was crucial for distinguishing between speakers and other text elements, but it was lost in the pre-processed data provided. By parsing the documents myself, I was able to retain and utilize this metadata, leading to more accurate speaker identification.

This approach ensured that I could reliably link each sentence to the correct speaker by using visual cues present in the original documents, which were otherwise overlooked in the pre-processed version.

### **2.1.3. Granular Control and Error Handling**

Custom parsing provided the ability to implement specific validation mechanisms and error handling tailored to the Knesset Corpus. This level of control ensured that edge cases and potential parsing errors were addressed promptly, resulting in higher accuracy and consistency in the processed data. This would have been challenging to achieve with a general-purpose parser like python-docx, which may not be equipped to handle the unique structure and content of the Knesset documents.

### **2.1.4. Enhanced Speaker Identification through Contextual Understanding**

By customizing the parsing process, I was able to better handle specific text structures and variations unique to the Knesset Corpus, such as Hebrew numbers and title variations (e.g., "Chairman" or "Minister"). Additionally, I could distinguish between actual speaker names and irrelevant text sections that might otherwise be mistakenly identified as speaker names. This contextual understanding of the documents allowed for a more accurate and thorough identification process, something that would be difficult to achieve using generic pre-processed data.

## **2.2. Implementation**

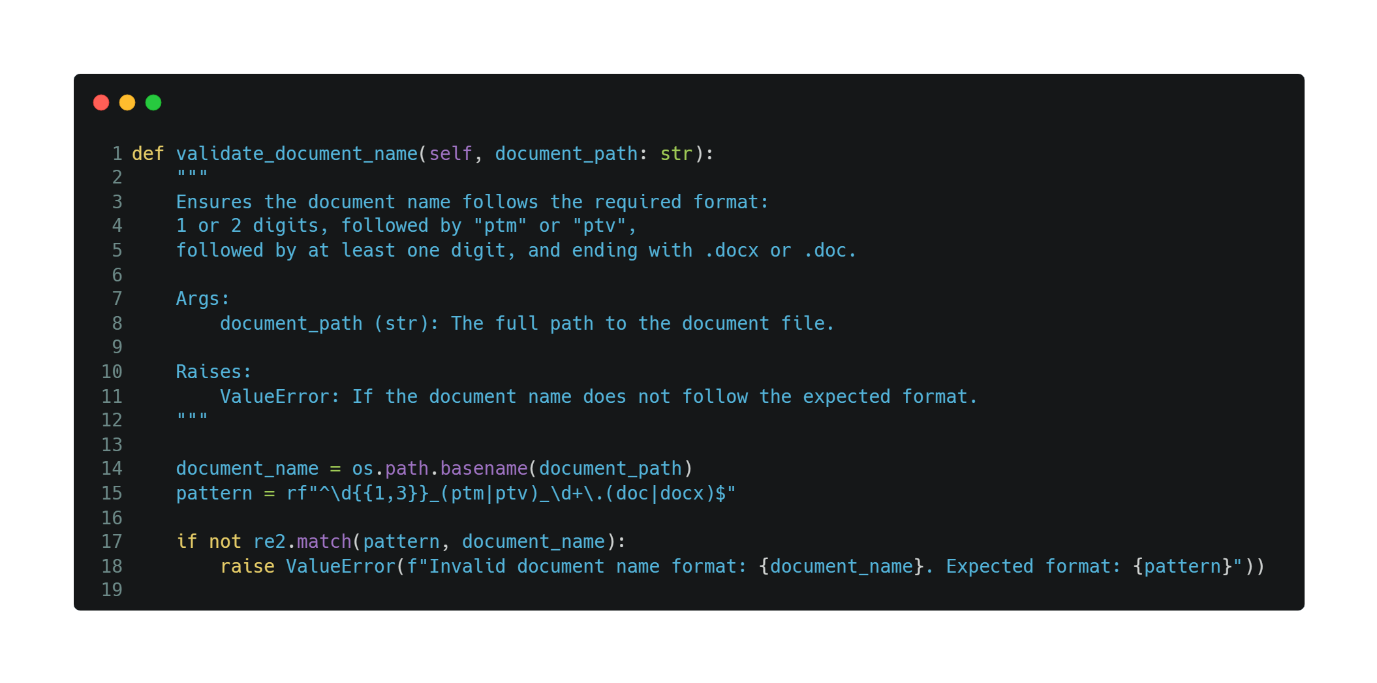
In the implementation, I developed separate parsing solutions tailored to both Windows and Linux environments to leverage the strengths of MS Word's COM object model on Windows and LibreOffice's headless mode on Linux, ensuring compatibility and functionality across different platforms. While macOS was considered for a potential implementation, it was ultimately decided that, given the complexities and limited use case for this specific project, the effort required to adapt the solution for macOS would outweigh the benefits at this stage. This decision allowed for a focused and efficient development process on the more critical platforms.

In addition, because of this implementation, an added bonus was that legacy **.doc files are supported directly**. Unlike with python-docx, where we would need to convert these files to .docx format before processing, my approach allows for seamless handling of both .doc and .docx files, streamlining the workflow and eliminating the need for additional conversion steps. This further enhanced the flexibility and robustness of the parsing solution.

### **2.2.0. Name Validation**

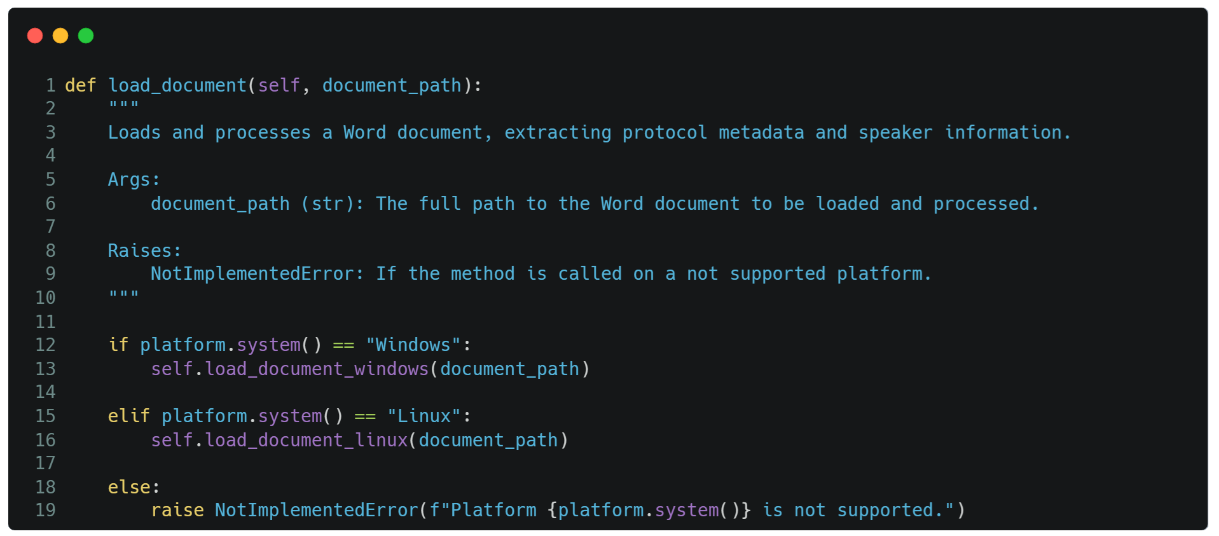
Ensuring that document names adhere to a specific format was crucial for maintaining consistency and preventing errors during processing. The required format for document names was defined as follows: one or two digits, followed by "ptm" or "ptv", then at least one digit, and ending with either .docx or .doc.

I implemented a validation function to enforce this naming convention. The function extracts the document name from the provided file path and checks if it matches the expected pattern using regular expressions. If the document name does not conform to the required format, the function raises a ValueError, alerting the user to the discrepancy.



In my implementation, I opted to use the Google **re2** library instead of Python's built-in **re** library because **re2** offers a significant performance advantage, operating in time complexity, whereas the standard Python **re** library can exhibit exponential time complexity in certain cases. This makes **re2** a more reliable choice for processing large datasets like the Knesset Corpus, where performance and efficiency are critical.

### **2.2.1. Load Document**



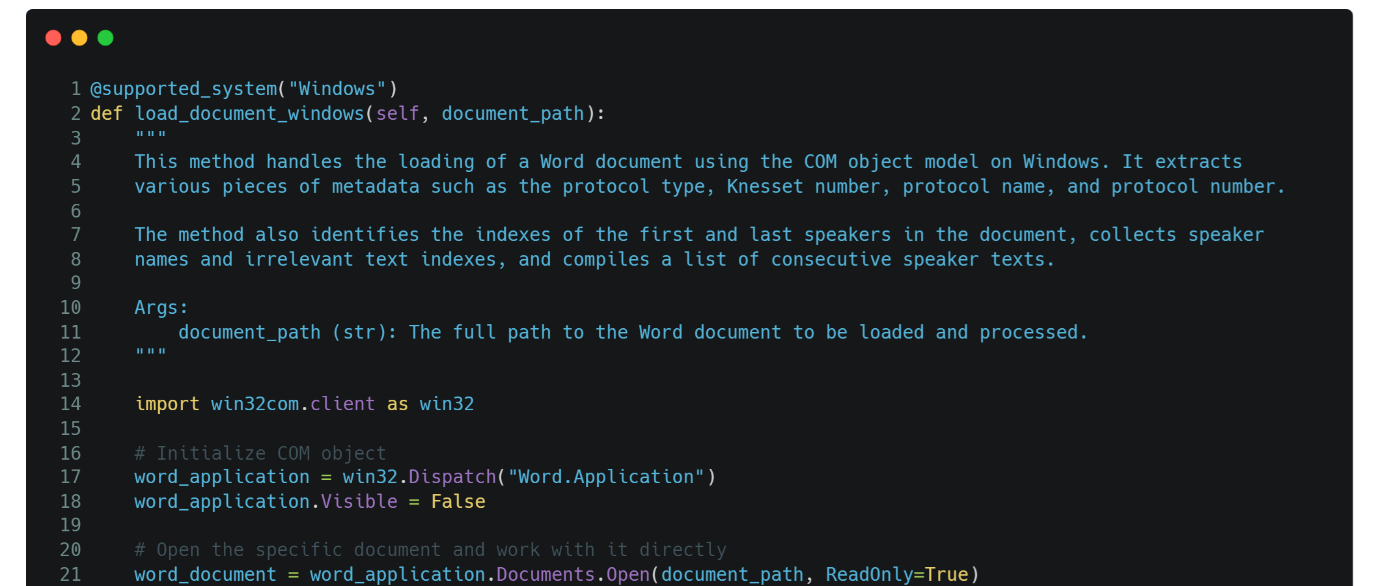
This method dynamically selects the appropriate document loading and processing implementation based on the operating system: on Windows, it uses the COM object model with Microsoft Word for direct interaction, while on Linux, it leverages LibreOffice in headless mode via a socket connection to extract the necessary metadata and speaker information.

Since the implementations for both operating systems share similar high-level concepts, I will explain them in parallel, highlighting the differences where they occur.

#### **2.2.1.1. Initialization and Setup**

* Both methods are decorated with **@supported\_system("Windows”)** or **@supported\_system("Linux”)** to ensure they execute only on their respective operating systems.

##### **2.2.1.1.1. Windows**



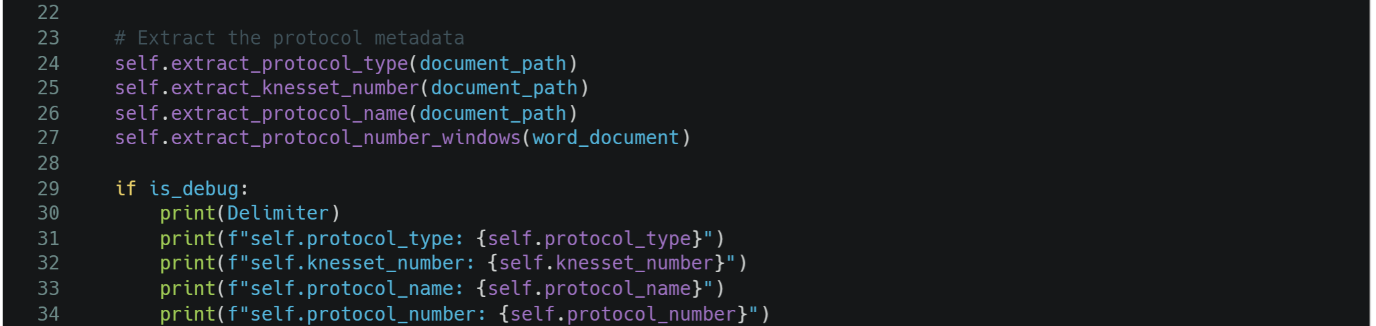
* It uses the COM object model to interact with Microsoft Word through the **win32com.client** library.
* Initializes a COM object (**word\_application**) for Microsoft Word and sets it to run invisibly (Visible = False).
* Opens the specified Word document in read-only mode.

##### **2.2.1.1.2. Linux**



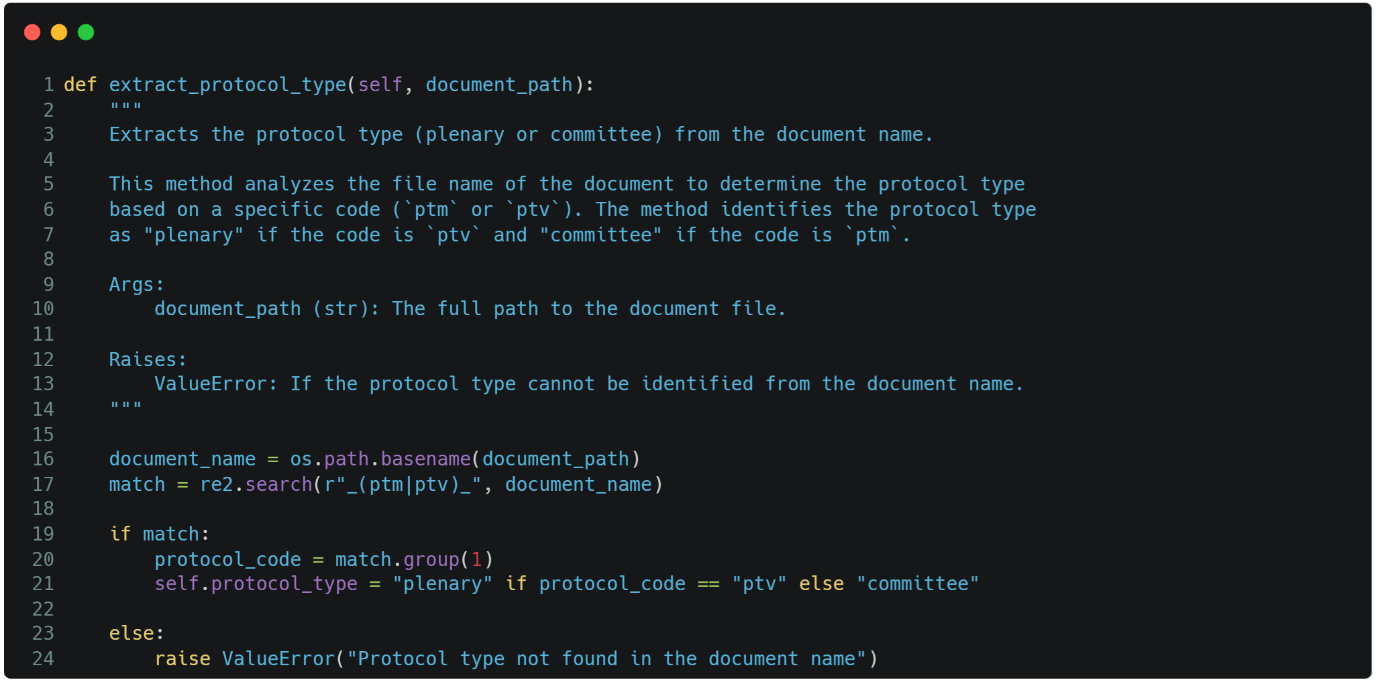
* It uses the UNO API to interact with LibreOffice in headless mode.
* Calls **KnessetProtocol.check\_libreoffice\_installed()** to ensure LibreOffice is installed.
* Calls **KnessetProtocol.ensure\_libreoffice\_headless()** to start LibreOffice in headless mode if it's not already running, storing the previous state in **was\_libreoffice\_headless\_open\_before**.
* Initializes the UNO component context and creates a UNO service manager to connect to the LibreOffice instance via a socket.
* Retrieves the desktop service from LibreOffice and prepares to open the document in a hidden mode.
* The method loads the specified LibreOffice Writer document using **desktop.loadComponentFromURL**, ensuring compatibility with Linux systems.

#### **2.2.1.2. Extract Metadata**



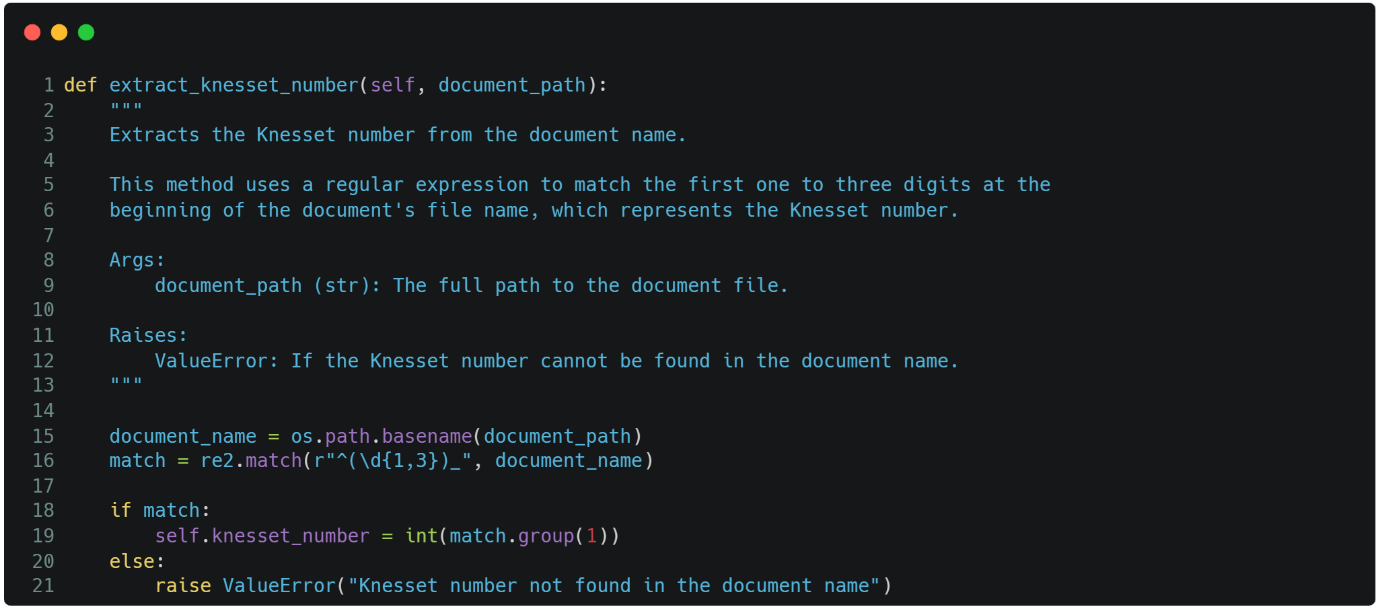
* The method calls several functions to extract and store various pieces of protocol metadata:
  + **self.extract\_protocol\_type(document\_path)** retrieves the type of protocol.
  + **self.extract\_knesset\_number(document\_path)** retrieves the Knesset number.
  + **self.extract\_protocol\_name(document\_path)** retrieves the protocol name.

##### **2.2.1.2.1. self.extract\_protocol\_type(document\_path)**

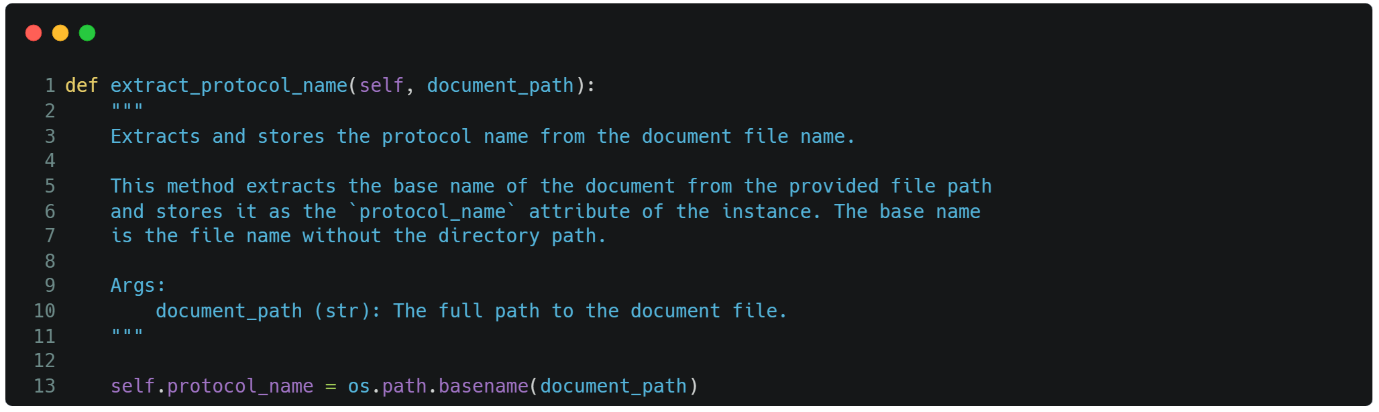


The **extract\_protocol\_type** method determines if the document is a "plenary" or "committee" protocol by analyzing its file name. It first isolates the file name using **os.path.basename(document\_path)**. Then, it uses **re2.search** to find the pattern "\_ptm\_" or "\_ptv\_". If "\_ptv\_" is found, **self.protocol\_type** is set to "plenary"; if "\_ptm\_" is found, it's set to "committee". If neither pattern is detected, the method raises a **ValueError**, ensuring that the protocol type is identified before proceeding.

##### **2.2.1.2.2. self.extract\_knesset\_number(document\_path)**

The **extract\_knesset\_number** method extracts the Knesset number from the document's file name by isolating the name using **os.path.basename(document\_path)** and then applying a regular expression to match one to three digits at the start. The pattern **r"^(\d{1,3})\_"** ensures that only the initial digits are captured. If a match is found, the digits are converted to an integer and stored in **self.knesset\_number**. If no match is found, the method raises a **ValueError**, indicating the Knesset number couldn't be identified.

##### **2.2.1.2.3. self.extract\_protocol\_name(document\_path)**



The **extract\_protocol\_name** method extracts the file name from the given document path and assigns it to the **protocol\_name** attribute. It uses **os.path.basename(document\_path)** to isolate the file name, removing any directory path, and stores this as the protocol's name.

##### **2.2.1.2.4. Windows**



First, the method initializes the **self.protocol\_number** attribute to **None** and defines a list of keywords that might indicate the presence of the protocol number, such as "הישיבה" and "פרוטוקול מס". Since Microsoft Word on Windows doesn't support regex or OR clauses in its search functionality, the method needs to handle each keyword separately.

Instead of iterating through the keywords in a loop—which could miss the earliest occurrence of any keyword depending on their order—the method sets up individual search ranges for each keyword. These ranges are then executed simultaneously to find all potential matches. After executing the search, it identifies the keyword that appears earliest in the document, ensuring the protocol number extraction is accurate and not biased by keyword order.

Once the earliest match is found, the method extends the search range to the end of the paragraph containing the keyword. It then extracts and attempts to parse the text immediately following the keyword, which should contain the protocol number. The method first tries to parse the number as a Hebrew numeral, and if that fails, it tries to parse it as a regular digit

If a protocol number is successfully extracted, the method concludes; otherwise, it raises a **ValueError**, indicating that no valid protocol number could be found.

The reason this approach is necessary on Windows is due to the limitations of Word's COM object model, which lacks advanced search features found in more modern text processing libraries. This method ensures that the correct protocol number is identified regardless of where it appears in relation to the keywords

##### **2.2.1.2.5. Linux**



The **extract\_protocol\_number\_linux** method is designed to extract the protocol number from a LibreOffice Writer document in a Linux environment. It works by searching for specific keywords, such as "הישיבה" or "פרוטוקול מס", which are likely to precede the protocol number in the document.

The method begins by initializing the **self.protocol\_number** attribute to **None**. It then creates a search descriptor within LibreOffice that allows for regular expression searches, enabling it to look for any of the specified keywords in the document. The search starts from the beginning of the document and continues until a match is found.

Once a keyword is located, the method captures the entire paragraph containing the match and extracts the text. It then attempts to identify the protocol number by parsing the text immediately following the keyword. The method first tries to parse the number as a Hebrew numeral, and if that fails, it attempts to parse it as a regular digit.

If a valid protocol number is found, the method stops searching; otherwise, it continues until all potential matches have been examined. If no protocol number can be extracted, the method raises a **ValueError**. This approach takes advantage of LibreOffice's regular expression search capability, making it more efficient and flexible compared to the Windows method.

#### **2.2.1.3. Identify first and last speaker indexes (Windows) / speaker ranges (Linux)**

##### **2.2.1.3.1. Windows**

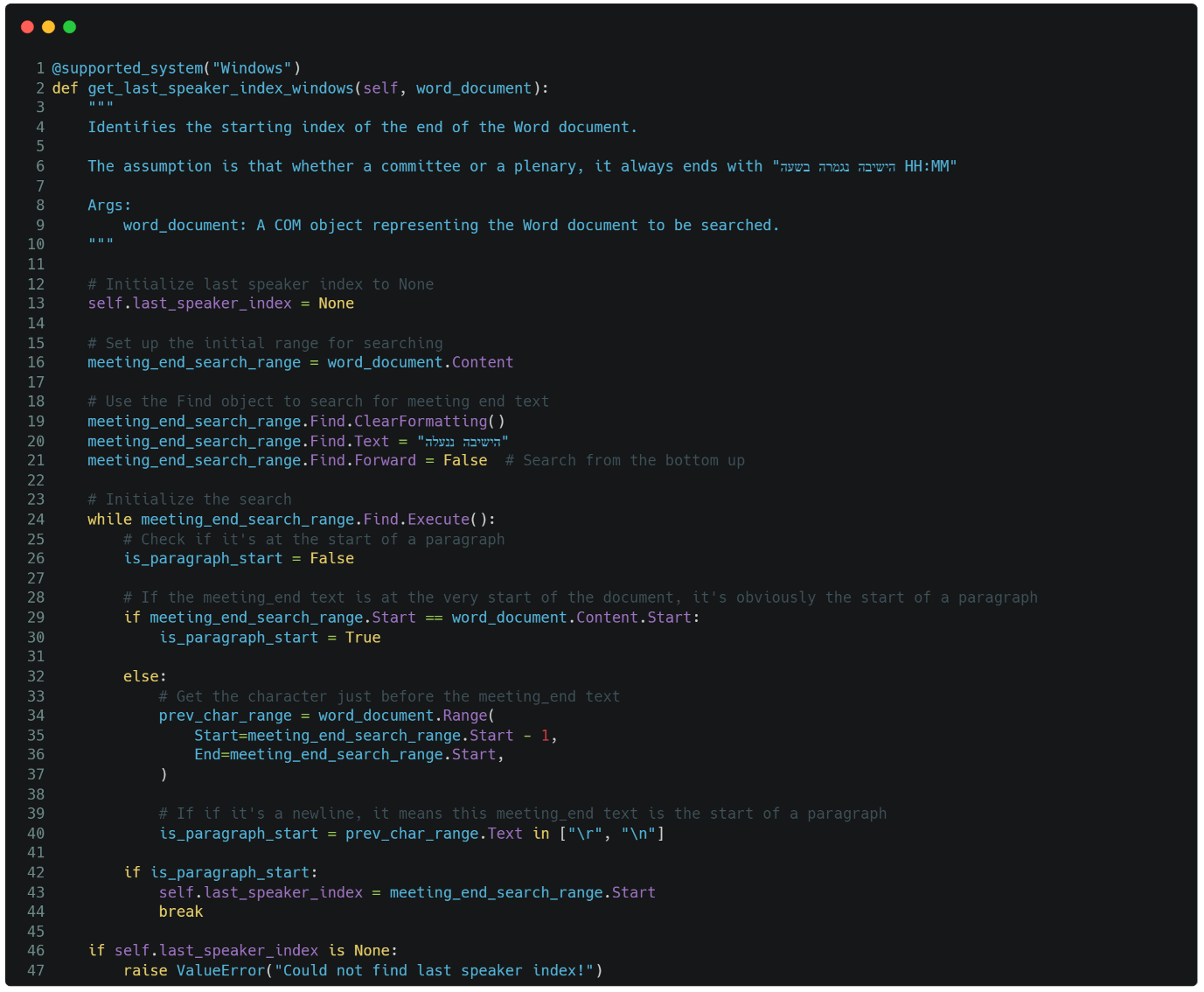


The **get\_first\_speaker\_index\_windows** method is designed to identify the starting position of the first speaker in a Word document. The method operates under the assumption that the first speaker is always the Knesset chairman (יו"ר). It utilizes Microsoft Word's COM object model to search through the document content, specifically looking for underlined text, which is often used to denote the chairman's name.

Here's how it works:

1. **Initialization:** The method initializes **self.first\_speaker\_index** to **None** to store the starting index of the first speaker once it is found.
2. **Search Setup:** It sets up a search range that encompasses the entire content of the Word document. The method then configures the Word Find object to search specifically for text that is underlined, which is a typical formatting style used for the chairman's name in Knesset protocols.
3. **Iterative Search:** The method enters a loop where it repeatedly searches for underlined text in the document:
   1. It retrieves the text that is underlined and checks if it is at the start of a paragraph. This is done by comparing the position of the underlined text with the start of the document or checking the character just before the underlined text.
   2. If the underlined text is immediately followed by a colon (:), which sometimes is not underlined itself, the method appends this colon to the underlined text.
   3. The method also checks if the text is centered, which might indicate that it is a title or something other than a speaker's name.
4. **Chairman Identification:** The method uses the **is\_chairman** static method to determine if the underlined text is likely referring to the Knesset chairman. This check ensures that the text is underlined, not centered, at the start of a paragraph, and matches specific patterns that indicate it is the chairman's name.
5. **Setting the First Speaker Index:** If the method successfully identifies the underlined text as referring to the chairman, it records the starting index of this text as **self.first\_speaker\_index** and breaks out of the loop.
6. **Error Handling:** If no appropriate underlined text is found that matches the criteria, the method raises a **ValueError**, indicating that it could not determine the first speaker's index.

This method systematically searches the Word document for the underlined name of the Knesset chairman, which is assumed to be the first speaker, and records the position where this name appears in the document. If it cannot find such a name, it raises an error.



The **get\_last\_speaker\_index\_windows** method is designed to identify the starting index of the final speaker or section in a Word document on a Windows system, with the assumption that the document ends with the phrase "הישיבה ננעלה בשעה HH:MM" ("The meeting was adjourned at HH:MM"). Here's how the method works:

1. **Initialization:** The method begins by setting **self.last\_speaker\_index** to **None**, which will later hold the starting index of the last speaker or the final relevant section of the document.
2. **Search Setup:** The method sets up a search range (**meeting\_end\_search\_range**) to cover the entire content of the Word document.
3. **Configuring the Search:** The method uses the Word COM object's **Find** feature to search for the phrase "הישיבה ננעלה" within the document. It clears any previous formatting settings to ensure that the search is only for this specific text. Additionally, the search is configured to run in reverse (from the bottom of the document to the top) by setting **Find.Forward = False**, as this phrase typically appears near the end of the document.
4. **Iterative Search:** The method then iterates through the document looking for occurrences of the phrase:
   1. For each occurrence found, it checks if the phrase appears at the start of a paragraph. This is determined by examining the character immediately before the found text. If this character is a newline (\r or \n), it indicates that the phrase begins a new paragraph.
   2. If the phrase is at the start of a paragraph, the method assigns the starting position of this phrase to **self.last\_speaker\_index**, marking it as the beginning of the final section or speaker's content in the document.
5. Conclusion: If the phrase "הישיבה ננעלה" is found at the start of a paragraph, the search stops, and the index is stored. However, if the method fails to find this phrase anywhere in the document, it raises a **ValueError**, indicating that it couldn't locate the expected final section.

This method scans the Word document from the end upwards, looking for the phrase "הישיבה ננעלה" to identify where the last relevant section or speaker begins. If found, it records the starting index; if not, it raises an error. This helps in delineating where the formal proceedings conclude, which is crucial for accurate text parsing and analysis.

##### **2.2.1.3.2. Linux**

The **get\_first\_speaker\_index\_linux** method is designed to identify the starting position of the first speaker, assumed to be the Knesset chairman (יו"ר), in a LibreOffice Writer document on a Linux system. Although named **first\_speaker\_index**, the method actually identifies the range within the document where the first speaker's text begins.

Here's how it works:

1. **Initialization:** The method initializes **self.first\_speaker\_index** to **None**, which will later store the starting point of the first speaker's range if found.
2. **Search Setup:** The method sets up a search descriptor within the LibreOffice Writer document to locate underlined text, which is typically used to denote the chairman's name. This descriptor includes properties for case-insensitive search and focuses specifically on text styles, particularly the underline style.
3. **Iterative Search:** The method begins searching the document using the defined search descriptor:
   1. It retrieves the underlined text that matches the search criteria.
   2. It checks if this underlined text is at the start of a paragraph, which could indicate it’s a speaker's name. This is determined by checking the character immediately preceding the underlined text to see if it is a newline character, which would suggest paragraph separation.
   3. The method then checks if the text immediately following the underlined text is a colon (:). If it is, the colon is appended to the underlined text because the full name of the speaker might include this punctuation.
4. **Chairman Identification:** The method checks if the underlined text corresponds to the Knesset chairman by using the **is\_chairman** function. This function evaluates whether the text is underlined, not centered, starts a paragraph, and matches certain patterns associated with the chairman's name.
5. **Recording the First Speaker Range:** If the method identifies the underlined text as belonging to the Knesset chairman, it records the starting position of this text as **self.first\_speaker\_index**, effectively marking the range where the first speaker begins.
6. **Error Handling:** If the method finishes searching the document without finding an appropriate match for the chairman, it raises a **ValueError**, indicating that it could not determine the first speaker's range.

The method scans through the LibreOffice document, looking for underlined text that likely represents the chairman's name. Once found, it records the range where this text begins, allowing subsequent processes to identify where the first speaker's content starts in the document. If it fails to locate such text, it raises an error to indicate the problem.

The **get\_last\_speaker\_index\_linux** method is designed to identify the starting index of the final section in a LibreOffice Writer document, which typically contains the text "הישיבה ננעלה בשעה HH:MM" ("The meeting was adjourned at HH:MM"). Although the method is named to find the last speaker's index, it actually locates the text range where the final relevant section begins.

Here's how it works:

1. **Initialization:** The method begins by setting **self.last\_speaker\_index** to **None**. This attribute will hold the starting index of the final section once it's found.
2. **Setting Up the Search:** A search descriptor is created to define the search parameters. The **SearchString** is set to "הישיבה ננעלה", the phrase that marks the end of the meeting. The search is configured to run backwards (**SearchBackwards = True**), meaning it starts from the end of the document and works its way up.
3. **Positioning the Cursor:** A text cursor (**search\_range**) is positioned at the end of the document. This ensures that the search begins at the bottom and moves towards the top.
4. **Executing the Search:** The method uses the **findNext** function to search for the specified text. It searches for the occurrence of "הישיבה ננעלה" as defined by the search descriptor.
5. **Text Analysis:** For each found occurrence, the method retrieves the text and checks whether it's at the start of a paragraph. To do this, it examines the character immediately before the found text:
   1. If the text is at the very start of the document, it's considered the start of a paragraph.
   2. If it's not at the start, the method checks if the previous character is a newline (\r or \n). If so, this confirms that the text is at the start of a new paragraph.
6. **Setting the Last Speaker Index:** If the text is found to be at the start of a paragraph, the method records the starting index of this text range in **self.last\_speaker\_index**. This indicates where the final section of the document begins.
7. **Stopping the Search:** Once the method identifies this index, it prints debug information (if debugging is enabled) and stops the search by breaking out of the loop.
8. **Validation:** If the method completes the search and does not find the desired text, it raises a **ValueError**, indicating that it couldn't locate the final section.

This method effectively identifies the range where the last relevant section of the document begins by searching for a specific closing phrase, ensuring that it is indeed at the start of a paragraph.

#### **2.2.1.4. Identify Speaker Names and Irrelevant Text**

##### 2.2.1.4.1. Windows



The **get\_speakers\_names\_indexes\_irrelevant\_text\_indexes\_windows** method is part of a system designed to process a Microsoft Word document, typically a Knesset protocol, and identify the names and locations of all speakers within the document. This method is crucial for accurately mapping dialogue to specific speakers and filtering out irrelevant text, such as titles or specific phrases that do not contribute to the speaker identification process.

Here's how it works:

1. **Initialization**: The method starts by initializing two lists: **speaker\_names\_indexes** to store the identified speaker names and their corresponding start and end indexes in the document, and **irrelevant\_text\_indexes** to store sections of text that are not relevant for speaker identification, such as titles or phrases like "קריאה" (a call).
2. **Setting Up the Search**: A search range is defined from the location of the first identified speaker to the last one in the document. The method configures the Microsoft Word **Find** object to search specifically for underlined text within this range. This is because speaker names are typically underlined in the documents being processed.
3. **Searching and Identifying Speakers**: The method iterates through the document, searching for each occurrence of underlined text within the specified range. For each underlined text found:
   1. It checks whether the text is at the start of a paragraph by looking at the character immediately preceding the text. If the character is a newline, or if the text is at the very start of the document, it is considered the start of a paragraph.
   2. It checks if the text is centered within the paragraph, which can be a characteristic of speaker names in certain documents.
   3. If the text ends with a colon (:) that is not underlined but belongs to the name, it appends this colon to the underlined text.
   4. The method then uses the **is\_speaker** function to determine if the underlined text is likely a speaker’s name. This function checks whether the text is underlined, at the start of a paragraph, and matches a specific regex pattern for speaker names.
   5. If identified as a speaker’s name, the method cleans the text using the **clean\_speaker\_name** function, which removes extraneous elements like party names or titles, and adds the cleaned name along with its start and end indexes to the **speaker\_names\_indexes** list.
4. **Identifying Irrelevant Text**: If the underlined text does not match a speaker's name but is still at the start of a paragraph, it is likely irrelevant text such as a title or the words "קריאה" or "קריאות". Such text is added to the **irrelevant\_text\_indexes** list. However, if the text is just a string of new lines or tabs, it is ignored.
5. **Validation**: After processing the entire search range, the method checks if any speaker names were identified. If no names were found, it raises a **ValueError**, indicating an issue with the speaker identification process.

This method systematically scans a Word document for underlined, paragraph-starting text to identify speaker names and filter out non-relevant sections, enabling precise speaker attribution in the processed document.

##### 2.2.1.4.2. Linux



The **get\_speakers\_names\_indexes\_irrelevant\_text\_indexes\_linux** method is designed to identify and catalog the names and locations of speakers, as well as irrelevant text sections, in a LibreOffice Writer document. This method specifically targets underlined text, which is typically used to indicate speaker names in the document, and checks whether this text is a speaker’s name or an irrelevant section like a title or a phrase such as "קריאה" (a call).

Here’s how it works:

1. **Initialization**: The method begins by initializing two lists: **speaker\_names\_indexes** for storing identified speaker names and their corresponding start and end positions in the document, and **irrelevant\_text\_indexes** for sections of text that are not relevant to the speaker identification process.
2. **Search Setup**: A search descriptor is created to search for underlined text within the document. The descriptor is configured to include styles in the search and to look specifically for text with a single underline.
3. **Search Execution**: The search is performed within the defined range, which spans from the **first\_speaker\_index** to the **last\_speaker\_index**. The method iterates through the document, searching for occurrences of underlined text.
4. **Text Evaluation**: For each piece of underlined text found:
   1. The method checks if the text falls within the specified range (between **first\_speaker\_index** and **last\_speaker\_index**).
   2. It determines if the text is at the start of a paragraph by examining the character immediately before it. If the text is at the beginning of the document or follows a newline character, it is considered the start of a paragraph.
   3. It checks if the text is centered in the paragraph, as this is another common characteristic of speaker names.
   4. If the text ends with a colon (":"), which may not be underlined but is part of the speaker's name, it appends this colon to the underlined text.
   5. The method then uses the **is\_speaker** function to determine if the underlined text is likely a speaker’s name. This function checks whether the text is underlined, at the start of a paragraph, and matches a regex pattern that typically corresponds to speaker names.
   6. If identified as a speaker's name, the method cleans the text using the **clean\_speaker\_name** function to remove extraneous elements like party names or titles, and then adds the cleaned name along with its start and end positions to the **speaker\_names\_indexes** list.
5. **Identifying Irrelevant Text**: If the underlined text does not match a speaker's name but is still at the start of a paragraph, it is likely irrelevant text such as a title or the words "קריאה" or "קריאות". Such text is added to the **irrelevant\_text\_indexes** list. However, if the text consists only of newlines or tabs, it is ignored.
6. **Validation**: After processing the entire range, the method checks if any speaker names were identified. If no names were found, it raises a **ValueError**, indicating that no speaker names could be detected within the specified range.

This method systematically scans a LibreOffice document for underlined, paragraph-starting text to identify speaker names and filter out non-relevant sections, ensuring accurate speaker attribution in the document.

#### **2.2.1.5. Generate Consecutive Speaker Texts**

##### 2.2.1.5.1. Windows



This code creates a list of consecutive speaker texts by iterating over the identified speaker names and their corresponding start and end indexes within the document. For each speaker, it checks if there are any irrelevant text sections between the end of the current speaker's text and the start of the next speaker's text. If irrelevant text is found, the text is extracted up to the start of the irrelevant section; otherwise, it goes up to the start of the next speaker's section.

The extracted text between these indexes is then cleaned by replacing line breaks, tabs, and other whitespace characters with a single space, and removing multiple consecutive spaces. The cleaned text is paired with the speaker's name and appended to the **speaker\_text\_consecutive** list, which stores the text associated with each speaker in sequence.

##### 2.2.1.5.2. Linux



This code creates a list of consecutive speaker texts by iterating over identified speaker ranges within a LibreOffice Writer document. Even though variables are named start\_index and end\_index, they actually represent text ranges, not simple numeric indexes.

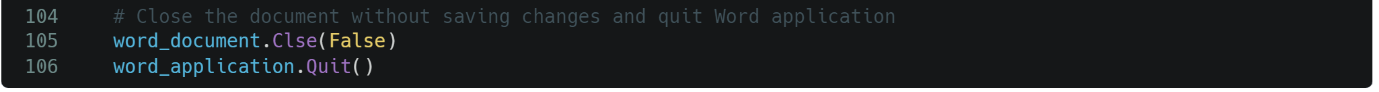
Here’s how it works:

1. **Text Range Comparison (\_\_get\_min\_text\_range)**:
   1. The function **\_\_get\_min\_text\_range** is defined to determine the earliest text range among a list of ranges. It compares each range with others to see which one starts before the majority of the others. The range with the highest score is considered the earliest and is returned.
2. **Iterating Over Speaker Ranges**:
   1. The main loop iterates through the identified speaker ranges (**speaker\_names\_indexes**), comparing the end of the current speaker’s range with the start of the next speaker’s range.
   2. It checks if there are any irrelevant text ranges (e.g., interruptions or titles) between the current and next speaker. If there are, it determines the earliest irrelevant text range that starts after the current speaker's end using **\_\_get\_min\_text\_range**.
   3. If irrelevant text is found, the end of the text to be extracted is set to the start of this irrelevant range; otherwise, it goes up to the start of the next speaker's range.
3. **Extracting and Cleaning Text**:
   1. A text cursor is created to extract the text between the current speaker's end and the determined end (either the next speaker's start or the irrelevant text's start).
   2. The extracted text is cleaned by replacing line breaks, tabs, and other whitespace characters with a single space, and removing multiple consecutive spaces.
   3. The cleaned text is then paired with the current speaker’s name and appended to **speaker\_text\_consecutive**, which holds all the speaker texts in sequence.

This approach ensures that the text associated with each speaker is accurately captured and cleaned, ready for further processing, such as sentence splitting.

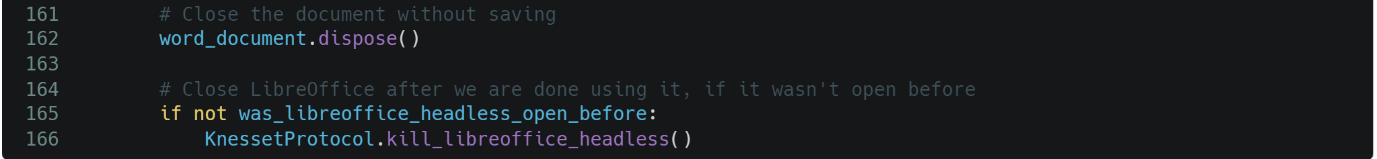
#### **2.2.1.6. Final Steps**

##### 2.2.1.6.1. Windows



This code closes the Word document without saving any changes (**Close(False)**) and then quits the Word application (**Quit()**), ensuring that the Word process is properly terminated.

##### 2.2.1.6.2. Linux



This code disposes of the LibreOffice document (**dispose()**) without saving, and then closes the LibreOffice headless process (**kill\_libreoffice\_headless()**) if it wasn't running before, ensuring no unnecessary processes are left open.

# **3. Metadata**

I got a csv containing detailed information about various Knesset Members along the years, looking like this:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PersonID | LastName | FirstName | GenderID | GenderDesc | Email | IsCurrent | LastUpdatedDate | DateOfBirth | PlaceOfBirth | YearOfAliyah | DateOfDeath | MotherTongue | Religion | Nationality | ReligiousOrientation | Residence | Sources | Notes |
| 2116 | א'-דאהר | אחמד | 251 | זכר |  | FALSE | ######### | ######### | ארץ ישראל |  | ######### |  | מוסלמי | ערבי |  | נצרת | https://he.wikipedia.org/wiki/%D7%90%D7%97%D7%9E%D7%93\_%D7%90-%D7%93%27%D7%90%D7%94%D7%A8; https://main.knesset.gov.il/mk/pages/MkPersonalDetails.aspx?MKID=129 | |
| 2117 | אבו-רביעה | חמאד | 251 | זכר |  | FALSE | ######### | ######### | ארץ ישראל |  | ######### |  | מוסלמי | בדואי |  |  | https://he.wikipedia.org/wiki/%D7%97%D7%9E%D7%90%D7%93\_%D7%90%D7%91%D7%95\_%D7%A8%D7%91%D7%99%D7%A2%D7%94; https://main.knesset.gov.il/mk/pages/MkPersonalDetails.aspx?MKID=180 | |
| 2118 | אבו-רוכן | לביב | 251 | זכר |  | FALSE | ######### | ######### | ישראל |  | ######### |  | דרוזי | דרוזי |  | עספיא | https://he.wikipedia.org/wiki/%D7%9C%D7%91%D7%99%D7%91\_%D7%97%D7%95%D7%A1%D7%99%D7%99%D7%9F\_%D7%90%D7%91%D7%95\_%D7%A8%D7%95%D7%9B%D7%9F; https://main.knesset.gov.il/mk/pages/MkPersonalDetails.aspx?MKID=233 | שם נוסף: חוסיין |
| 30071 | אבו מערוף | עבדאללה | 251 | זכר |  | FALSE | ######### | ######### | ישראל |  |  |  | דרוזי | דרוזי |  | ירכא | https://he.wikipedia.org/wiki/%D7%A2%D7%91%D7%93%D7%90%D7%9C%D7%9C%D7%94\_%D7%90%D7%91%D7%95\_%D7%9E%D7%A2%D7%A8%D7%95%D7%A3; https://main.knesset.gov.il/mk/pages/MkPersonalDetails.aspx?MKID=909 | |
| 23642 | אבו עראר | טלב | 251 | זכר | tabuarar@knesset.gov.il | FALSE | ######### | ######### | ישראל |  |  |  | מוסלמי | בדואי |  | ערערה | https://he.wikipedia.org/wiki/%D7%98%D7%9C%D7%91\_%D7%90%D7%91%D7%95\_%D7%A2%D7%A8%D7%90%D7%A8; https://main.knesset.gov.il/mk/pages/MkPersonalDetails.aspx?MKID=857 | |
| 30628 | אבו רחמון | ניבין | 250 | נקבה | ozer\_aburahmoun@knesset.gov.il | FALSE | ######### | ######### | ישראל |  |  |  | מוסלמי | ערבי |  | ריינה, בועיינה-נוג'ידאת | https://he.wikipedia.org/wiki/%D7%A0%D7%99%D7%91%D7%99%D7%9F\_%D7%90%D7%91%D7%95\_%D7%A8%D7%97%D7%9E%D7%95%D7%9F; https://main.knesset.gov.il/mk/pages/MkPersonalDetails.aspx?MKID=971 | |
| 30751 | אבו שחאדה | סמי | 251 | זכר | sabuschada@knesset.gov.il | TRUE | ######### | ######### | ישראל |  |  |  | נוצרי | ערבי |  | תל אביב | https://he.wikipedia.org/wiki/%D7%A1%D7%90%D7%9E%D7%99\_%D7%90%D7%91%D7%95\_%D7%A9%D7%97%D7%90%D7%93%D7%94; https://main.knesset.gov.il/MK/APPS/mk/mk-personal-details/1031 | מתגורר ביפו |

I manually went over each member, and checked it it’s notes contained more information about his name – another name, or a nickname which is commonly used.

I then created a table in the following format:

| **person\_id** | **last\_name** | **first\_name** | **last\_name\_2** | **first\_name\_2** | **last\_name\_3** | **first\_name\_3** | **gender** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **i64** | **str** | **str** | **str** | **str** | **str** | **str** | **str** |
| 2116 | "א'-דאהר" | "אחמד" | null | null | null | null | "זכר" |
| 2117 | "אבו-רביעה" | "חמאד" | null | null | null | null | "זכר" |
| 2118 | "אבו-רוכן" | "לביב" | null | "חוסיין" | null | null | "זכר" |
| 30071 | "אבו מערוף" | "עבדאללה" | null | null | null | null | "זכר" |
| 23642 | "אבו עראר" | "טלב" | null | null | null | null | "זכר" |

This table contains 3 columns for both last\_name and first\_name. Later on, I will check all possible combinations of last\_name\_{x} and first\_name\_{y} . This way, even if a person has a second name, or a nickname, we consider it in our comparisons and not just push it all to the same string (or ignore it).

Then, I ran the same clear names logic from above:

| **person\_id** | **last\_name** | **first\_name** | **last\_name\_2** | **first\_name\_2** | **last\_name\_3** | **first\_name\_3** |
| --- | --- | --- | --- | --- | --- | --- |
| **i64** | **str** | **str** | **str** | **str** | **str** | **str** |
| 2116 | "א דאהר" | "אחמד" | null | null | null | null |
| 2117 | "אבו רביעה" | "חמאד" | null | null | null | null |
| 2118 | "אבו רוכן" | "לביב" | null | "חוסיין" | null | null |
| 30071 | "אבו מערוף" | "עבדאללה" | null | null | null | null |
| 23642 | "אבו עראר" | "טלב" | null | null | null | null |

I saved this in **Parquet** format. I explain the reasons for using the **Parquet** format later on.

# **4. Matching speakers to actual Knesset members**

TODO: Explain how I matched each speaker to the Knesset Member in the metadata

# **5.** **Splitting to sentences**



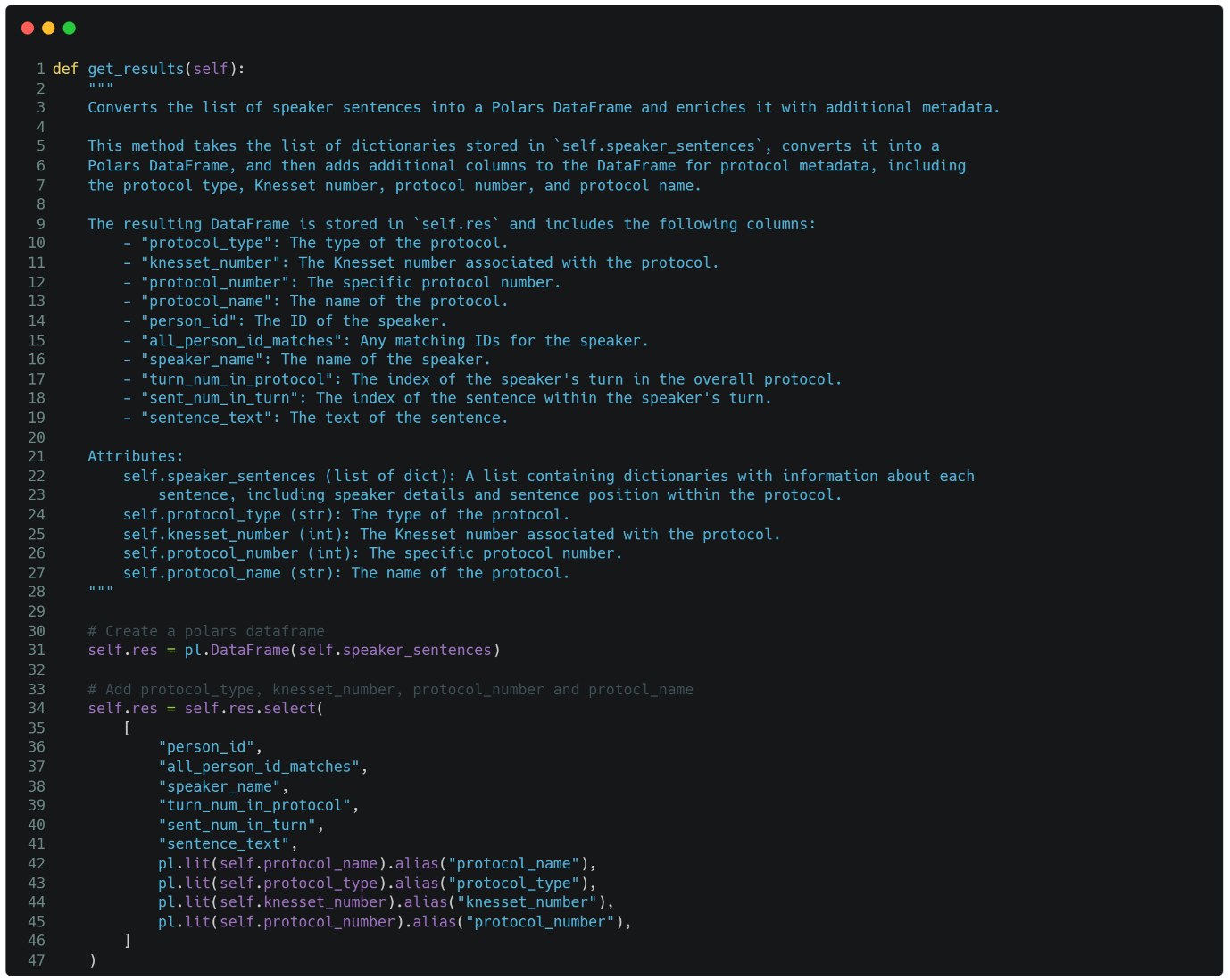
In the **split\_text\_to\_sentences** method, the NLTK library's **PunktSentenceTokenizer** is utilized to divide the text spoken by each speaker into individual sentences. The method starts by initializing this tokenizer, which is well-suited for recognizing sentence boundaries in a variety of texts based on punctuation and capitalization.

The method then iterates through a list that contains the text spoken by each speaker during their turn in a protocol. For each turn, the tokenizer splits the full text into sentences by identifying where one sentence ends and another begins, typically at punctuation marks like periods or question marks.

Once the sentences are identified, the method creates a dictionary for each sentence. This dictionary includes the speaker's name, their unique ID, any potential matching IDs, the sentence's text, the turn number in the overall protocol, and the sentence's position within that turn. These dictionaries are then collected in a list called **self.speaker\_sentences**.

This process effectively breaks down each speaker's contributions into individual sentences, making it easier to analyze or further process the data at a more granular level.

# **6. Get Results**



The **get\_results** method is responsible for converting the processed list of speaker sentences into a structured data format using the Polars library, which is a powerful and efficient DataFrame library for Python. This method enriches the data with additional metadata about the protocol, such as its type, the associated Knesset number, the specific protocol number, and the protocol name.

Here’s how it works:

**Conversion to DataFrame:** The method begins by taking the list of dictionaries stored in **self.speaker\_sentences** and converting it into a Polars DataFrame. Each dictionary in the list represents a sentence spoken by a Knesset member, along with associated metadata such as the speaker's name, their ID, and the sentence's position within the protocol.

**Enriching with Metadata:** After creating the initial DataFrame, the method adds additional columns that include the protocol metadata. Specifically, it uses the **pl.lit()** function from Polars to add constant values (i.e., the protocol metadata) across all rows in the DataFrame. These columns include:

* **protocol\_name**: The name of the protocol.
* **protocol\_type**: The type of the protocol (e.g., plenary or committee).
* **knesset\_number**: The Knesset number associated with the protocol.
* **protocol\_number**: The specific protocol number.

**Column Selection:** Finally, the method selects the relevant columns to be included in the final DataFrame. This step ensures that the DataFrame contains all the necessary information: speaker details, sentence metadata, and protocol-related data.

The resulting DataFrame is stored in **self.res**, which can be further used for analysis, storage, or reporting. This method effectively consolidates all relevant information into a single, easily manageable DataFrame.

**Why Polars?** I opted to use the Polars library over Pandas for several key reasons:

1. **Performance:** Polars is significantly faster than Pandas, sometimes up to 100 times faster, particularly when handling large datasets. This performance boost is crucial for processing extensive data, such as the Knesset Corpus.
2. **Scalability:** Polars is designed to handle much larger datasets than Pandas can efficiently manage, making it a better choice for tasks involving big data.
3. **Familiar Syntax:** Polars offers a syntax that is similar to Spark/SQL, which aligns well with the tools and workflows I am accustomed to in my job. This familiarity allows for a smoother and more intuitive experience when working with the data.

By using Polars, I can ensure that the data processing is both fast and efficient, while also leveraging a syntax that is powerful and easy to work with for complex data manipulation tasks.

# **7.** **Output Format**

The **write\_results** method is responsible for saving the processed data, stored in a Polars DataFrame, to disk in the Parquet format while ensuring proper partitioning using PyArrow. This method is critical for efficiently storing and organizing large datasets.

Here’s how it works:

**Conversion to PyArrow Table:** The method starts by converting the Polars DataFrame **self.res** into a PyArrow Table. This step is essential because PyArrow provides robust functionality for writing Parquet files, including advanced features like partitioning.

**Explicit Schema Definition:** An explicit schema is defined using PyArrow, specifying the data types for the columns **protocol\_type**, **knesset\_number**, and **protocol\_number**. Defining the schema ensures that the data is correctly interpreted and stored according to its intended structure.

**Writing to Parquet Format:** The method then writes the PyArrow Table to the specified **processed\_data\_path** in Parquet format. It uses PyArrow's **write\_dataset** function to handle this task, ensuring that the data is partitioned correctly on disk according to the specified schema. This partitioning is crucial for optimizing data retrieval and storage efficiency.

**Why Parquet over JSON:** Parquet is chosen as the storage format over JSON for several key reasons:

1. **Efficiency and Performance:** Parquet is a columnar storage format, which means that it stores data by columns rather than by rows. This structure allows for more efficient data compression and faster query performance, especially when working with large datasets. In contrast, JSON is a text-based format that stores data in a row-wise manner, which can lead to larger file sizes and slower performance.
2. **Compression:** Parquet files are highly compressed, often leading to significant savings in storage space. JSON, being a text format, typically results in larger file sizes, especially for complex datasets.
3. **Schema Evolution:** Parquet supports schema evolution, meaning that you can add new columns or modify the schema over time without disrupting existing data. JSON, on the other hand, is more rigid and doesn't naturally support schema evolution, making it less flexible for growing datasets.
4. **Better for Big Data:** Parquet is designed for big data scenarios and integrates well with distributed processing frameworks like Apache Spark or Elastic. This makes it a preferred choice in data engineering and analytics pipelines.

**Why PyArrow over Polars' Native write\_parquet:** I opted to use PyArrow's **write\_dataset** instead of Polars' native **write\_parquet** function because PyArrow handles disk partitioning more effectively. While Polars is excellent for in-memory data manipulation, PyArrow provides more advanced features for disk I/O, particularly when it comes to partitioning data into subdirectories based on specific columns.

Partitioning data on disk allows for better organization and faster access during read operations, as it enables the system to skip irrelevant partitions when querying the data. PyArrow’s **write\_dataset** function is designed to handle this partitioning robustly, ensuring that the data is stored in a way that maximizes efficiency.

The data processed by the **write\_results** method is saved inside the **data/processed** directory. The method ensures that the data is partitioned into subdirectories based on three key columns: **protocol\_type**, **knesset\_number**, and **protocol\_number**. This means that the data is stored in a hierarchical folder structure, where each level of the directory corresponds to a value in one of these columns.

For example, the structure might look something like this:



This partitioning strategy is beneficial because it allows for efficient data retrieval. When you query the data, the system can skip entire folders (partitions) that are irrelevant to your query, significantly reducing the amount of data that needs to be scanned and improving query performance.

The method ultimately ensures that the processed data is stored in a highly efficient, organized, and query-friendly format, making it ready for further analysis or use in data pipelines.

# **8. Testing**

TODO: Explain how I tested

# 9. References

* **The Knesset Corpus: An Annotated Corpus of Hebrew Parliamentary Proceedings** – [Gili Goldin](https://arxiv.org/search/cs?searchtype=author&query=Goldin,+G) (1), [Nick Howell](https://arxiv.org/search/cs?searchtype=author&query=Howell,+N) (2), [Noam Ordan](https://arxiv.org/search/cs?searchtype=author&query=Ordan,+N) (2), [Ella Rabinovich](https://arxiv.org/search/cs?searchtype=author&query=Rabinovich,+E) (3), [Shuly Wintner](https://arxiv.org/search/cs?searchtype=author&query=Wintner,+S) (1) ((1) Department of Computer Science, University of Haifa, Israel, (2) IAHLT, Israel, (3) School of Computer Science, The Academic College of Tel-Aviv Yaffo, Israel)  
  [**https://arxiv.org/abs/2405.18115**](https://arxiv.org/abs/2405.18115)