Оглавление

[Введение 2](#_Toc196319757)

[Документация модуля для парсинга docx 3](#_Toc196319758)

[Библиотеки 3](#_Toc196319759)

[Открытие файла 3](#_Toc196319760)

[Работа с параграфами 4](#_Toc196319761)

[Работа с таблицами 6](#_Toc196319762)

[Элементы файла типа xml 8](#_Toc196319763)

[Обработка и форматирование данных 9](#_Toc196319764)

[Получение всего текста из файла 9](#_Toc196319765)

[Присоединение титульного названия таблицы 13](#_Toc196319766)

# Введение

Данная документация создана для дальнейшего администрирования дополнительного модуля для СИП (Система интеллектуального поиска). Модуль позволяет обрабатывать документ формата docx, в котором корректно преобразует внутренние объекты файла в необходимый вид для LLM.

# **Документация модуля для парсинга docx**

## **Библиотеки**

Для начальной работы потребуются следующий список библиотек, представленный на листинге 1.

Листинг 1 — Используемые библиотеки.

import docx

from docx import Document

from docx.text.paragraph import Paragraph

from docx.table import Table

from docx.oxml.text.paragraph import CT\_P as omxl\_paragraph

from docx.oxml.table import CT\_Tbl as omxl\_table

from docx.oxml.section import CT\_SectPr as omxl\_section

import regex

Их можно установить, используя файл requirements.txt, через команду pip install -r requirements.txt. Используются следующие версии библиотек:

* python-docx==1.1.2;
* regex==2024.11.6

## **Открытие файла**

Далее, необходимо открыть файл, реализуется с помощью функции *open\_docx,* которая на вход принимает путь к файлу формата docx. Реализация представлена на листинге 2.

Листинг 2 — Открытие файла формата docx.

def open\_docx(path):

return Document(path)

На рисунке 1 показан результат работы листинга 2.



Рисунок 1 — Открытие файла docx

Как можем заметить, что у нас возвращается объект нашего документа, следовательно, он обладает следующими, для нас полезными, свойствами и атрибутами:

* Получение только параграфов, т.е. абзацы, разделенные переносом на другую строку, где каждый параграф есть объект;
* Получение только таблиц, где каждая таблица есть объект;
* Получение всех элементов файла в формате xml.

Стоит более подробно описать каждый из пунктов.

## **Работа с параграфами**

На листинге 3 будет представлено получение только тех элементов, которые являются параграфами.

Листинг 3 — Получение параграфов

def get\_paragraphs(document):

return document.paragraphs

В результате возвращается список объектов *docx.text.paragraph*. Результат представлен на рисунке 2.

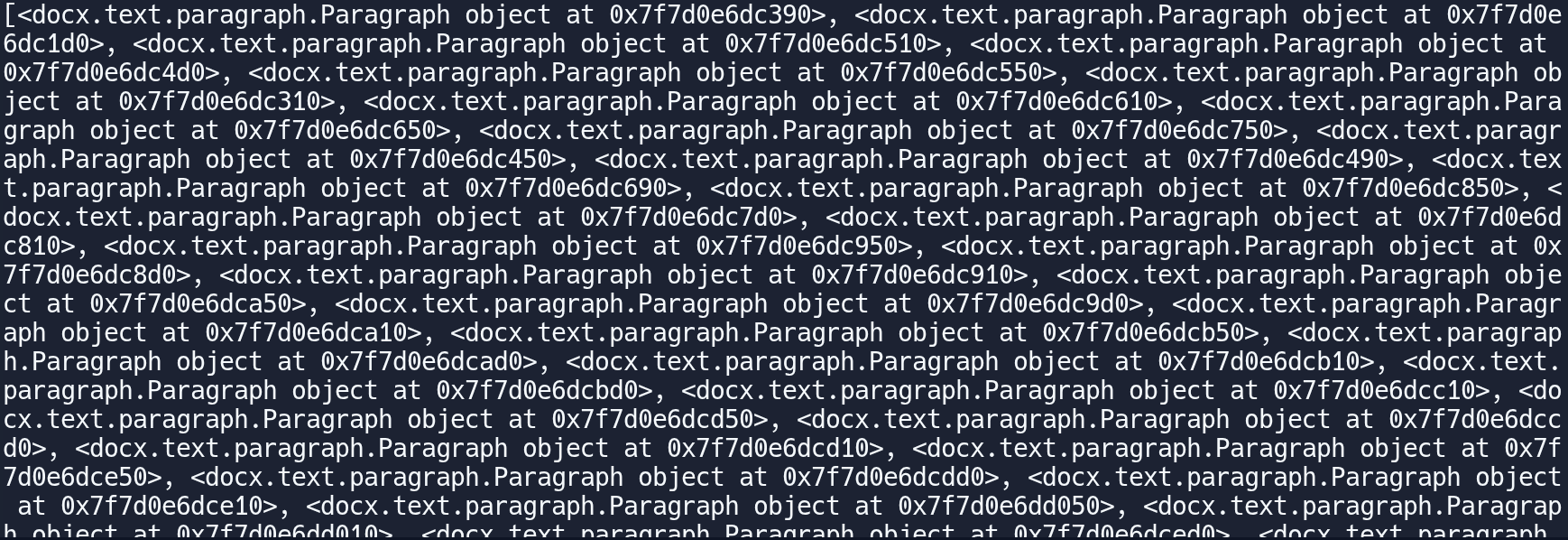


Рисунок 2 – Список объектов параграфов

Зная, что параграф другими словами — абзац, а абзац — набор слов, то у MS Word, как и в большинстве текстовых редакторах, присутствуют определенные свойства, такие как:

* Курсив;
* Жирный шрифт;
* Подчеркивание.

Это одни из основных, полный перечень можно узнать в документации MS Word.

В случае поставленной задачи парсинга, необходим сам текст, анализировать свойства не нужно. Получение текста представлено на листинге 4.

Листинг 4 — Получение текста из объектов *docx.text.paragraph*.

def get\_text(paragraph):

return paragraph.text

def iter\_paragraphs(paragraphs):

for paragraph in paragraphs:

print(get\_text(paragraph))

iter\_paragraphs(paragraphs)

На рисунке 3 будет представлен текст во входном документе, на рисунке 4 будет представлен результат получение данного текста.

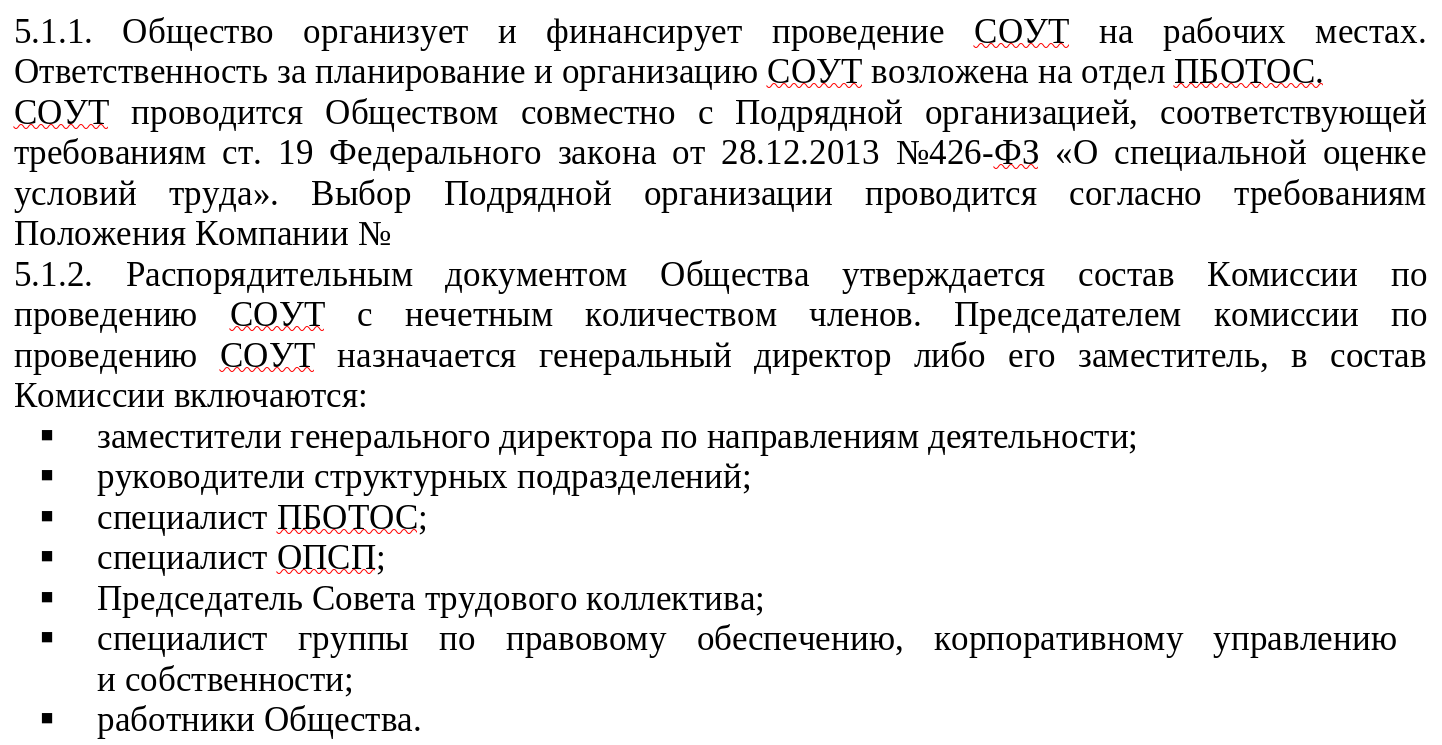


Рисунок 3 — Входной текст в документе

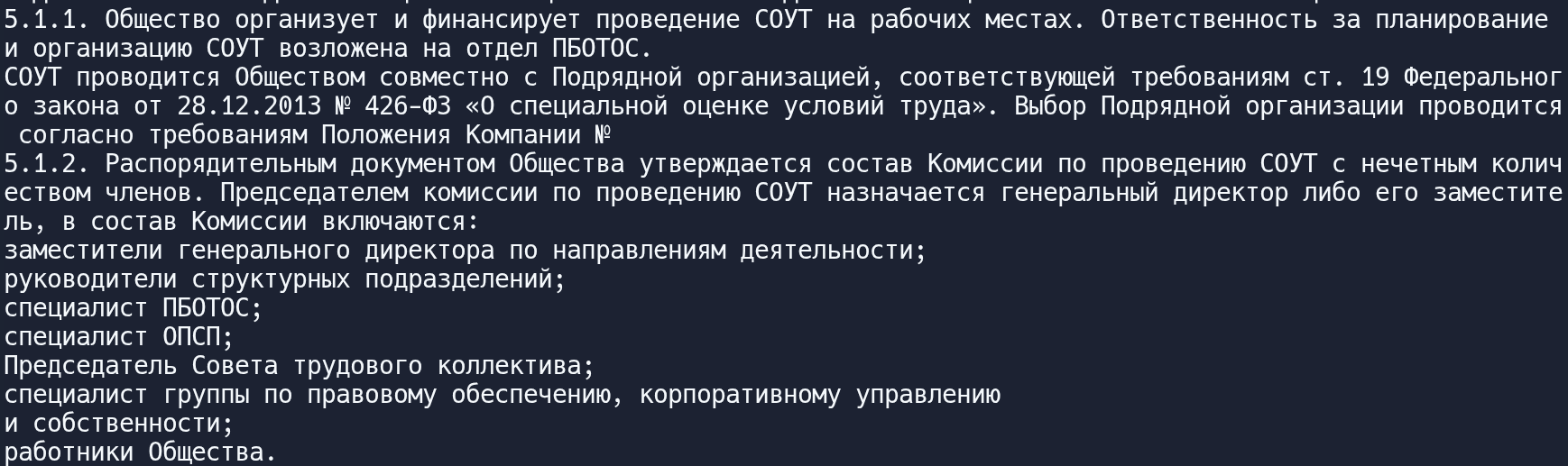


Рисунок 4 — Результат обработанного текста

## **Работа с таблицами**

Одна из специфичных задач — корректная обработка таблиц. Получение таблиц, как и параграфов, не составляет особой сложности. На листинге 5 представлено получение таблиц.

Листинг 5 — Получение списка таблиц.

def get\_tables(document):

return document.tables

tables = get\_tables (document)

print(tables)

Тут тоже возвращается список объектов, только уже *docx.table*. Результат представлен на рисунке 5.

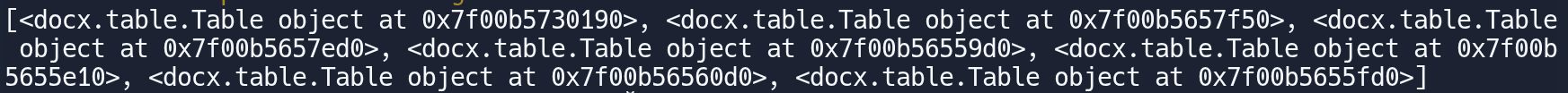


Рисунок 5 — Список объектов таблиц

Как таковых свойств таблиц замечено не было, поэтому остается только представить в читабельном виде. Способ получения представлен на листинге 6.

Листинг 6 — Читабельное представление таблиц.

def iter\_table(table):

for row in table.rows:

\_row = []

for cell in row.cells:

\_row += [cell.text]

print(\_row)

# взяли четвертую

# таблицу из списка

table = tables[3]

iter\_table(table)

Результат работы алгоритма представлен на рисунке 7. На рисунке 6 будет отображена таблица, которая находится во входном файле.

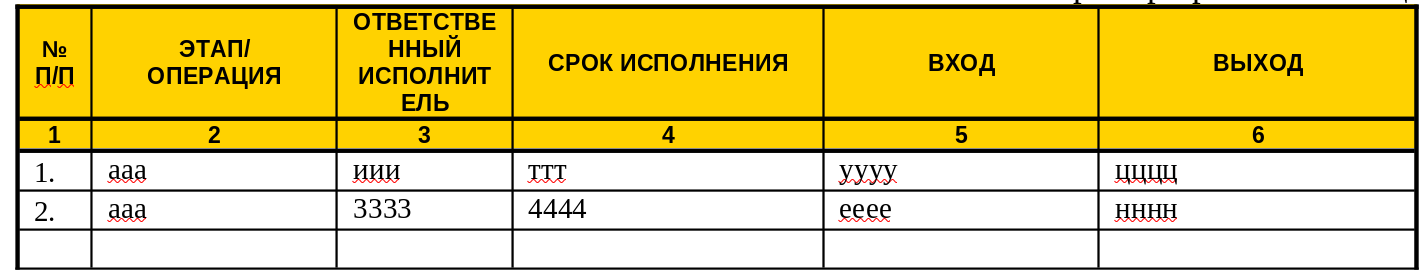


Рисунок 6 – Входная таблица, находящиеся в документе

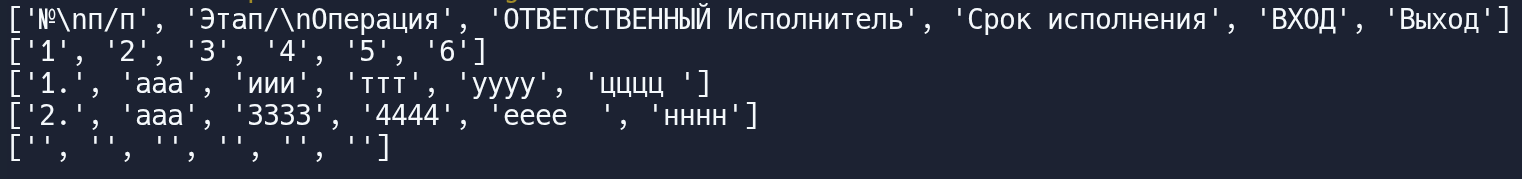


Рисунок 7 – Результат полученной преобразованной таблицы

Как можно было заметить, что при простой организации таблицы все обрабатывается корректно, но при разделении/объединение колонки/строки, происходит следующий результат, который показан на рисунках 8 — 11



Рисунок 8 – Входная таблица с объединением строк

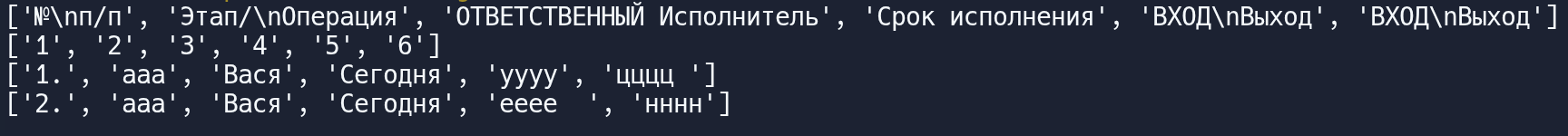


Рисунок 9 – Результат обработки с объединением строк



Рисунок 10 — Входная таблица с разделением столбцов

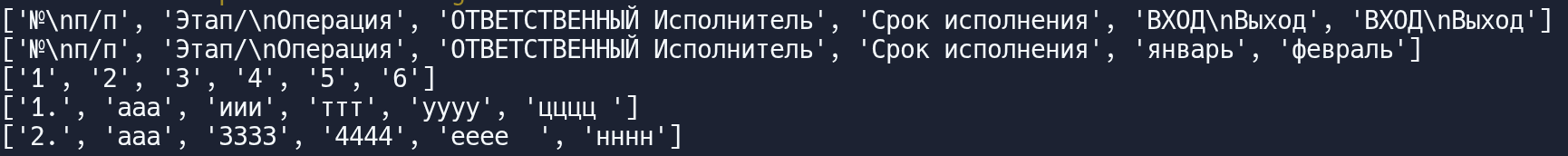


Рисунок 11 — Результат обработки с разделением столбцов

Из приведенных выше примеров наблюдается следующая зависимость — при объединении/разделение элементы повторяются, что и логично с точки зрения внутренней структуры xml. Это как раз хорошо повлияет для будущей поставленной задачи. Более детальная обработка таблиц будет показаны в разделе „Обработка и форматирование данных“.

## **Элементы файла типа xml**

MS Word по своей структуре представляет собой xml. Если до этого можно было получить отдельно параграфы и таблицы, то через элементы xml, можно последовательно получать каждый из этих элементов. Но не стоит забывать о том, что по мимо параграфов, таблиц в файле могут находиться другие объекты, например, как верхний/нижний колонтитул. Поэтому, необходимо учитывать только текст и таблицы. Данная выборка представлена в листинге 7.

Листинг 7 — Формирование основного тела файла.

def get\_body(document):

body = []

for element in document.element.body:

if isinstance(element, omxl\_paragraph) or isinstance(element, omxl\_table):

body += [element]

return body

body = get\_body(document)

print(body)

Результатом получим список из объектов xml параграфов и таблиц, как это показано на рисунке 12.



Рисунок 12 — Тело файла

## **Обработка и форматирование данных**

После того, как получи все необходимые данные их необходимо между собой объединить. Под этим подразумевается следующие:

* Преобразовать каждый объект в текст;
* Присоединить к таблице название таблицы.

### **Получение всего текста из файла**

Обработать объекты параграфа, как и было сказано в разделе „Получение таблиц“, не составляет труда. Куда больше идей потребует преобразовать таблицу в читабельный вид. На рисунках 7, 9, 11 пока преобразовали только в строки списков и если же захотим их соединить, то получим следующее (рисунок 13).

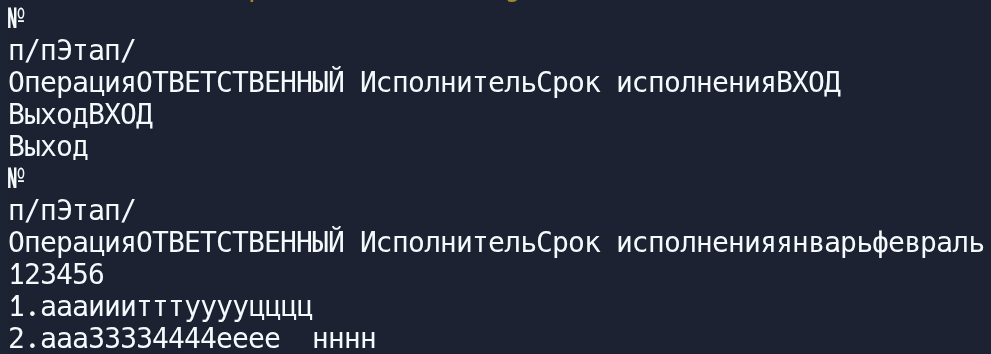


Рисунок 13 — Соединение элементов таблицы

Из изображения видно, что какие-то строки и столбцы съехали, да и не понятно, как они между собой связаны, поэтому необходимо сделать определенную *чистку* данных, т.е. удалить лишние отступы, пробелы и т.п. и добавить некий разделитель, как это отображено на рисунке 14.

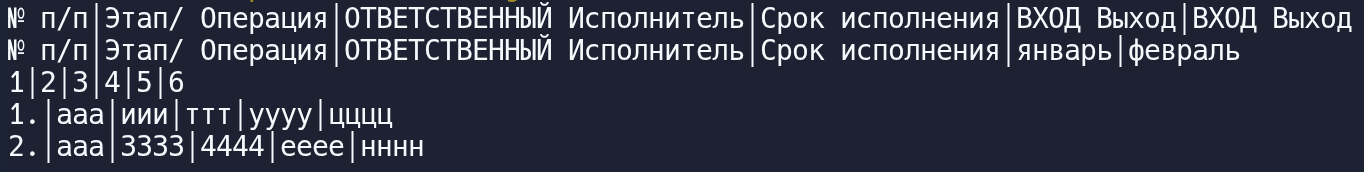


Рисунок 14 — Чистка данных для таблицы

Так на много лучше, но остается до сих пор не читабельно из-за разной ширины. Следовательно, для каждого столбца необходимо найти элемент, который имеет максимальную длину и выровнять по центу. Алгоритмы представлен в листинге 8 — 9.

Листинг 8 — Поиск максимального по ширине элемента столбца.

def lenght\_every\_column(table, rows, columns):

dict\_lenght\_columns = {}

for i in range(columns):

maxim\_lenght = 0

for j in range(rows):

maxim\_lenght = max(maxim\_lenght, len(table[j][i]))

dict\_lenght\_columns[f'столбец\_{i}'] = maxim\_lenght + 9 if maxim\_lenght % 2 else maxim\_lenght + 8

return dict\_lenght\_columns

Листинг 9 — Форматирование столбцов, учитывая ширину.

def format\_columns(table, rows, columns):

dict\_lenght\_columns = lenght\_every\_column(table, rows, columns)

for i in range(columns):

format\_space = dict\_lenght\_columns[f'столбец\_{i}']

for j in range(rows):

table[j][i] = f'{table[j][i]:^{format\_space}}'

Получаем следующий вид таблицы (рисунок 15).

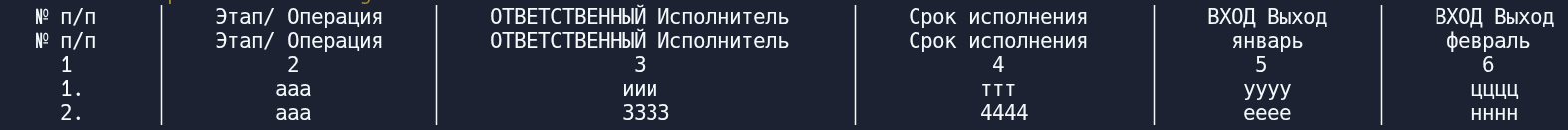


Рисунок 15 — Читабельный вид таблицы

Теперь, когда основная работа выполнена, немного добавим стилистики — разделим на ячейки. Реализация представлена в листинге 10.

Листинг 10 – Разделитель строк на ячейки.

def sep\_rows(row):

return f"{'─' \* len(row)}"

В результате получаем следующее (рисунок 16).

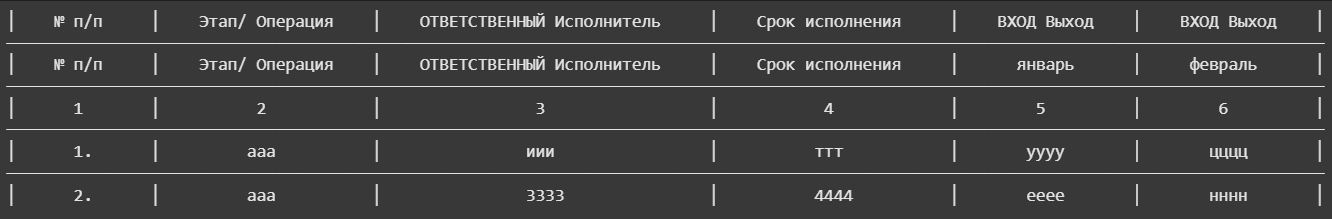


Рисунок 16 — Добавление разделителя

Но, как можно заметить линии между собой не соединены. Зная, про разделяющий символ между элементами, будем находить его индекс и при создании разделителя стоки на данное место будем помещать другой разделяющий символ, это показано в листинге 11.

Листинг 11 — Поиск индексов для замены на разделяющий символ.

def index\_sep\_rows(row, columns):

index\_sep = []

count = 0

while count != columns:

index = row[1:].index('│')

index\_sep += [index + 1]

row = row[index + 1:]

count += 1

return index\_sep

Теперь, перейдем к замене, тогда модифицируем листинг 10.

Листинг 10 – Разделитель строк на ячейки (модификация).

def sep\_rows(index\_sep):

return f"\n├{'┼'.join(list(map(lambda x: '─' \* (x - 1), index\_sep)))}┤\n"

Получаем следующее (рисунок 17).

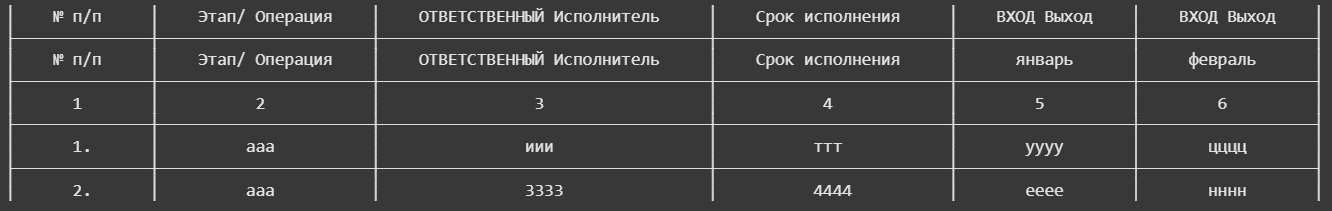


Рисунок 17 — Модификация разделителя

Остается добавить верхний и нижний разделитель, отличаются только в элементах разделения. Они представлены на листингах 12 – 13.

Листинг 12 – Разделитель строки для начальной строки

def sep\_begin\_row(index\_sep):

return f"├{'┬'.join(list(map(lambda x: '─' \* (x - 1), index\_sep)))}┤"

Листинг 13 – Разделитель строки для последней строки

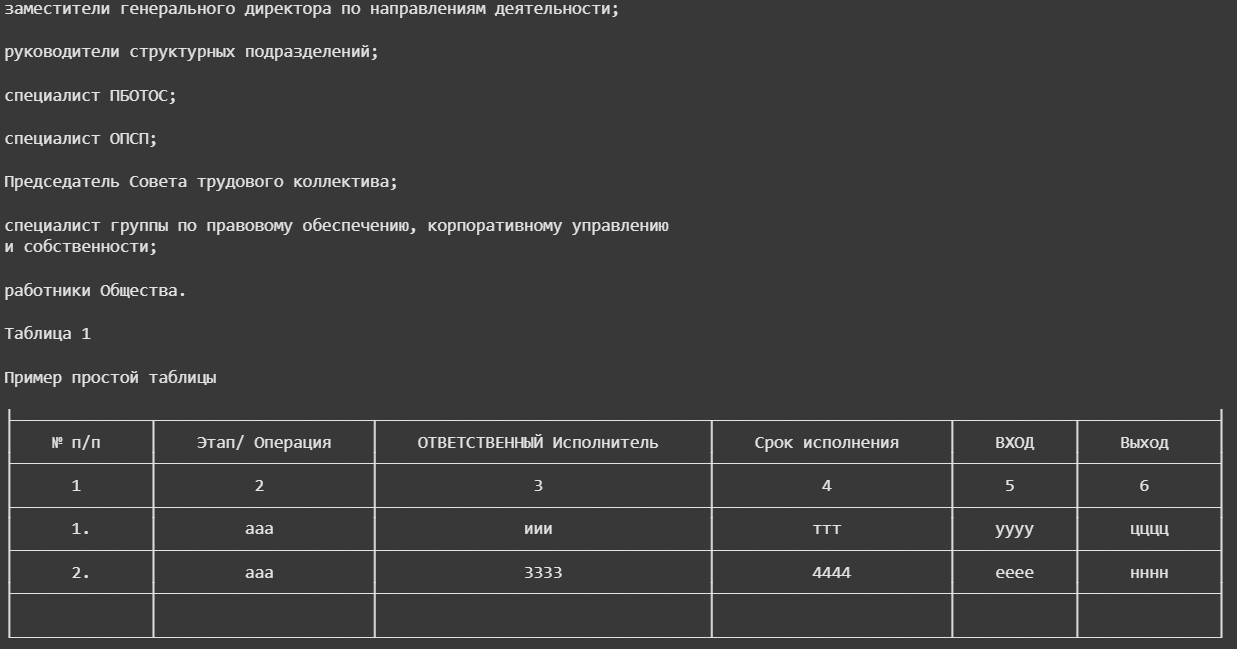
def sep\_last\_row(index\_sep):

return f"└{'┴'.join(list(map(lambda x: '─' \* (x - 1), index\_sep)))}┘"

В результате получаем почти полноценную таблицу (рисунок 18).

Рисунок 18 — Полноценная читабельная таблица

Соединим текст и таблицы, полученный результат на рисунке 19.

Рисунок 19 — Объединенный текст и таблица

### **Присоединение титульного названия таблицы**

Как видно из рисунка 19, что титульное название таблицы и сама таблица – отдельно друг от друга, но это можно исправить – присоединив к верхней части таблице.

Тогда, для начала потребуется скорректировать тело файла таким образом, что собрать все то относящиеся к названию таблицы, сконвертировать в тип xml и добавить в уже новое тело функции, тогда получится, что название таблицы будет стоять сразу перед таблицей и когда мы встречаем таблицу, то берем предыдущий объект и присоединяем к таблице. Пример идеи можно увидеть на рисунке 20.

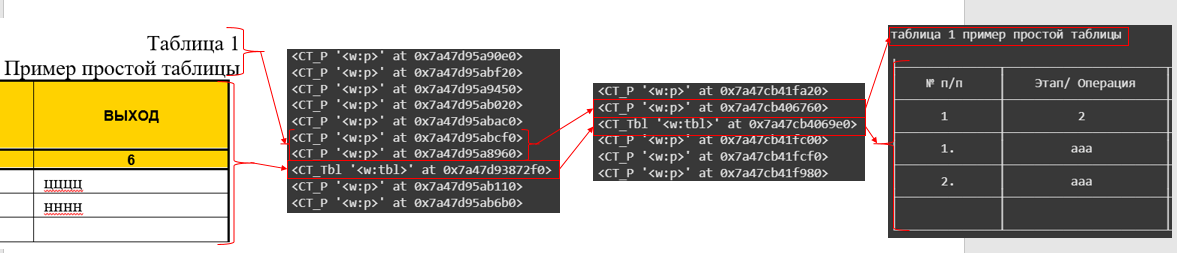


Рисунок 20 — Идея преобразования тела файла

На листинге 14 будет представлен алгоритм преобразования тела файла.

Листинг 14 – Преобразование тела файла.

def correct\_body(body, document):

\_body = []

index\_title = None

for i, element in enumerate(body):

if isinstance(element, omxl\_paragraph):

text = convert\_omxl\_paragraph\_to\_text(element, document)

if text:

if regex\_title(text):

index\_title = i

elif not index\_title:

\_body += [element]

else:

if index\_title:

title = convert\_list\_omxl\_paragraph\_to\_text(body[index\_title:i], document)

\_body += [convert\_text\_ to\_text\_omxl\_paragraph(title), element]

index\_title = None

return \_body

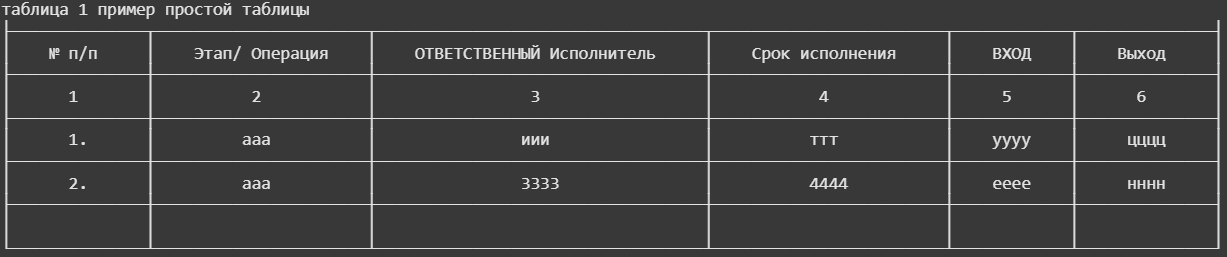
После того, как сформировали новое, корректное тело файла, нам необходимо находить название таблицы. В листинге 14 данная функция была упомянута *regex\_title*. Через регулярное выражение ищет, что это таблица. Реализация представлена на листинге 15.

Листинг 15 – Поиск табличного название через регулярное выражение.

def regex\_title(text):

return regex.findall(r'таблица\s[0-9]{1,}', text) != []

Теперь, когда все необходимые элементы находятся друг с другом можем их соединить, как показано на рисунке 21.

Рисунок 21 — Соединенное название таблицы и таблицы

Теперь приступим к форматированию. Могут возникать ситуации, когда название таблица куда больше чем сама таблица после форматирования, поэтому это необходимо учитывать. Зная ширину каждого столбца можно сравнить с длиной название таблицы, после добавить разницу к каждой ширине столбца. Реализация представлена на листинге 16.

Листинг 16 – Дополнительное форматирование столбцов.

def add\_space(dict\_lenght\_columns, title, columns):

lenght\_title = len(title)

if sum(list(dict\_lenght\_columns.values())) + columns <= lenght\_title:

return lenght\_title // columns

return 0

Тогда модифицируем листинг 9.

Листинг 9 – Форматирование столбцов, учитывая ширину (модификация).

def format\_columns(table, rows, columns):

dict\_lenght\_columns = lenght\_every\_column(table, rows, columns)

space = add\_space(dict\_lenght\_columns, title, columns)

for i in range(columns):

format\_space = dict\_lenght\_columns[f'столбец\_{i}'] + space

for j in range(rows):

table[j][i] = f'{table[j][i]:^{format\_space}}'

Осталось отформатировать название таблицы с учетом длины строки. Реализуется на листинге 17.

Листинг 17 – Форматирование названия таблицы.

def sep\_title(title, lenght):

return f"╭{lenght \* '─'}╮\n│{title:^{lenght}}│"

В итоге получаем обработанный, отформатированный текст файла (рисунок 22).

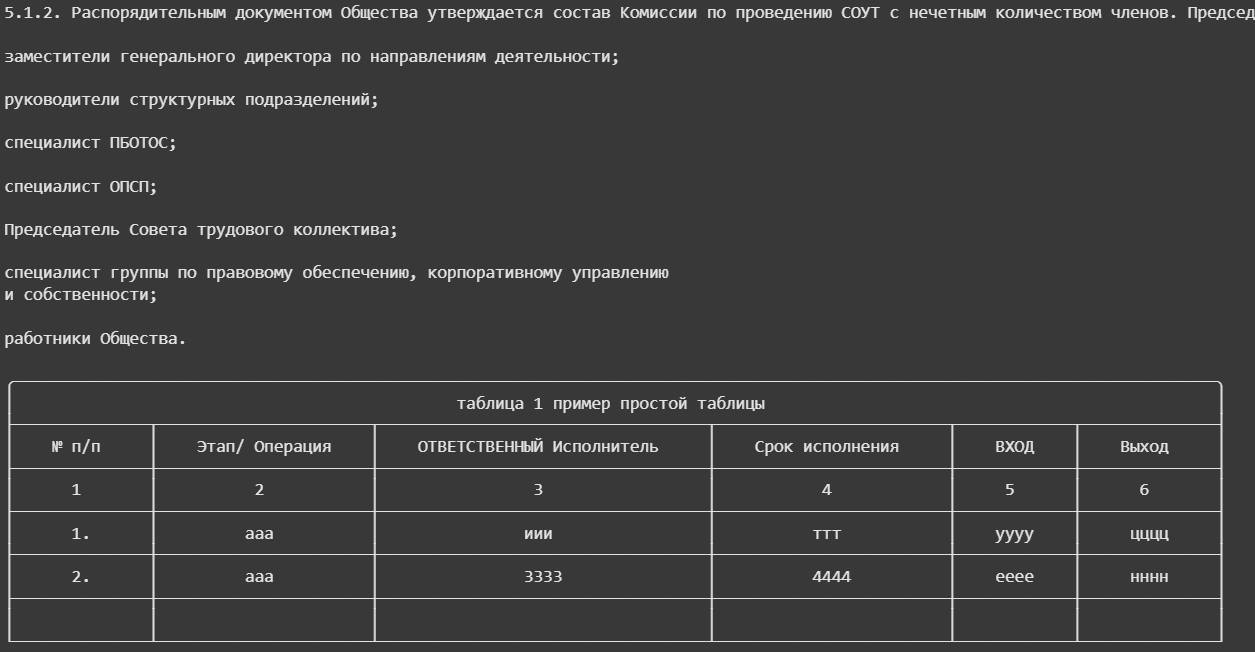


Рисунок 22 — Часть обработанного файла