PDFParser代码架构

背景

要实现的功能

将上市公司定期报告中的表格读取出来，然后整理成一个一个的Pandas DataFrame。上市公司定期报告总体上是高度格式化的。也就是说，它们的目录、内容结构是非常相似的。但是，也并不是所有的报告都具有相同的section和subsection，在section的具体名称方面，不同上市公司的定期报告会有较大差异。在少数情况下，section的前后顺序也可能不一致。

我们要读取的PDF文件，文件质量一般都比较高，不是从图片或扫描件转换为PDF文件的，而应当是从原始的文字编辑器（如word）中直接转为PDF的。目前的代码使用 `pdfplumber` 包来提取PDF中的表格。提取到的表格准确率是非常高的。之所以仍然要写很多代码，**核心目的是：给提取出来的表格命名，即要确定提取出来的表格对应的是上市公司定期报告哪个部分的哪个表。**

`pdfplumber` 能实现两个主要功能：1、把每一页PDF中的文字（含汉字）提取出来，返回一个很长的字符串；2、把每一页的每个表提取出来，每一个表返回一个list of list。

在“PdfRecognition/原始数据“文件夹下，有几个PDF文件，代码是以处理这些PDF文件为例的。通过查看这些PDF文件就会发现，上市公司定期报告中的表格，有如下特点，并给 `pdfplumber` 提取表格制造了困难：

1. 绝大多数表格都没有“表头”。但是，可以根据紧邻表格上面的文字标题，确定这是什么表格。不过，这行文字标题与表格之间可能仍然有少量的，但数量不等的文字。 `pdfplumbe`

`r` 并没有返回表格所在的坐标位置，所以很难判断表格“紧邻上面”的那一行文字。

2. 从PDF文件打印出的效果来看，换行符只在该出现的时候出现。但是，`pdfplumber` 在读取文字时，会给汉字中间添加大量的换行符。因此，无法根据换行符的位置来判断是否在PDF文件中真的换了行。
3. 一个表格可能会跨页。一页也可能有多个表格。`pdfplumber` 只会依次标明某一页PDF文件的第几个表格，但是如何从内容和逻辑上判断跨页的表格，这是一个难点。
4. 很多表格会有“合并单元格”的情况。这给 `pdfplumer` 识别表格造成了特殊的困难。`pdfplumber` 提取文字，并输出为字符串时，是根据页面上一整行来输出的，而不会根据表格内的换行进行处理（表格中的内容如果有汉字，也会存在同样问题）。也就是说，`pdfplumber` 识别文字时，无法处理由于合并单元格造成的逻辑上的换行。
5. 除了上述问题之外，还会有别的坑，需要在写代码的过程中逐渐发现和解决。

总体设计思路

`pdfplumber` 提取表格本身，没有太多问题。问题的关键是，把每一个提取出的表格，与PDF文件内容的逻辑上的结构建立联系。要建立联系，核心是确定每一个表格与其上、其下文字的相对位置。

由于 `pdfplumber` 并没有返回文字和表格的真实坐标（即没有距上下左右多少厘米），所以判断相对位置，是通过寻找特定的字符串（更常见的是特定的字符block）来实现的。这就大量地使用了 `re` 模块。

目前已经写好的Parser类及其方法

目前写了一个类，来专门处理PDF表格提取的问题。这个类命名为 `Parser`，它的定义在ParsingRegularReports01文件夹下。这个类的基本定义在“__init__.py”文件中，其他方法分布在各个代码文件中，一般来说，方法的名字和代码文件的名字是一致的。

main.py文件及Parser类的整体结构

main.py是程序的入口。它的运行结构依次（这个结构顺序是有意义的，不是随便先运行一个方法，再运行另外一个方法的）是：

1. 初始化这个类。其中，建立了年报常见的section结构（半年报、季度报告的结构与年报是不同的，此处仅建立了年报的结构，其他定期报告的结构仍待建立）。年报的section结构又分为两部分，一种是一级标题，在代码中被称为standard_sections；另一种是财务报告这个一级标题下的各个二级标题，在代码中被称为standard_fin_subsections。详见“__init__.py”文件。

2. `read_in_pdf()`：把一个.pdf文件读入到内存中，并抽取出每一页的文字和表格。特别要注意建立的`self.all_pages`和`self.all_tables`这些属性。这里要注意，在读取和保留下来的每页的文字和表格时，对一些没用的字符做了剔除，之后在寻找时，都是使用这种已经剔除了无用字符后的字符串进行查找的。

3. `find_header()`：找到所有的页眉。但是要注意，并不是每一页PDF文件都有页眉。而且，目前只能假定一个PDF文件中的页眉是保持一致的。未来也许会发现某些PDF文件的页眉会不一致（这多半不是刻意的，而且报告的撰写者的笔误）。之所以要找到页眉，是为了便于判断每一页的正文是从哪里开始的。

4. `find_toc()`：找到目录页，并提取其中的一级标题和对应的页码。一般来说，上市公司的定期报告中没有二级标题。运行这个函数后，会更新`self.possible_tocs`这个属性，从中可以看出标题和页码是否正确。

5. `find_sections_pages()`：继续在正文中查找前一步找到的一级标题及其页码，进一步判断是否有错误。运行这个函数后，会更新`self.section_titles`这个属性，用于检查是否正确。

6. `self.compare_standard_toc()`：将找到的目录与标准目录对照，看是否一致。执行这一步的目的是，考虑到有可能不同的PDF在一级标题的文字上略有差异，而最终将表格对应到PDF文本内容时，希望所有的PDF提取结果都使用同一套一级标题体系。这一步会显示，是否完全一致，如果不一致，也许需要进一步手工处理。也许，这一步可以用算法，把近似的标题对应起来?

7. `self.find_fin_report_structure()`：寻找财务报告这个一级标题下的二级标题。其思路和做法与寻找一级标题类似。可以查看`self.fin_subsections_compared_std`这个属性。

8. `self.find_positions_of_all_tables()`：开始寻找每一个表的位置。注意，每个table在哪个page，这在`pdfplumber`提取时就已经一一对应了。但是，这个table在这个page的具体哪个位置，是这个函数要解决的问题。“位置”是指从第几个字符开始，到第几个字符结束。如何寻找每个表的位置，是个比较复杂的算法。

1. 它的中心逻辑是，把一个表里所有的字符都找出来，然后看这个字符在这个表中出现了几次。然后在本页的全部字符中逐个查找，看有没有一个一整块字符串，其中每个字符出现的次数与表中每个字符出现的次数完全相符。如果完全相符就找到了这个表。
2. 这个算法不完美。它的缺点是运算起来时间长。当然，目前来看，这个时间仍然是可以接受的。
3. 这个算法的另一个缺点是，它可能找出多个位置。例如，如果这个表之前或之后的一行，与这个表的第一行或最后一行文字一模一样，那么就可能出现多个位置；另外，如果一个表很小，没有多少字，那么这个函数也可能找到多个位置。
4. 之前曾经写过别的算法，但是问题更多。例如，使用这个表的第一行或第一个单元格的文字来寻找这个表的开始位置，这种做法的致命缺点是，当第一行有合并单元格时，`pdfplumber`提取的字符顺序与我们在文本中看到的顺序不一致，会导致无法找到这个表格的位置。
5. 当出现一个表有多个位置时，还会进一步判断哪些位置是错误的。到底在哪里做的这个判断，我记不得了。思路是：如果找到了所有表的位置，而`pdfplumber`又明确地指出了哪个表在前，哪个表在后，那么就可以根据这个顺序，来判断一个表的位置是不是错误了。可以把所有找到的表的位置组合起来，这会产生很多很多种组合，然后看哪种组合，能够确保所有表的先后顺序是正确的，就把这个组合保留下来。目前看，通过这种做法，能够最终找到一个组正确的表的位置。

9. `self.remove_one_table(139)`：极少数情况下，某些表是冗余的，或存在其他错误的。此时，手工确认后，需要删除这个表格。可以使用这个函数。出现冗余表格的一种情境是，不知道什么原因，`pdfplumber`会把一个table中的某个单元格识别成一个表，这个表就是冗余的，应当删除。

另外，tools文件夹里，有一些上述函数共用的方法，这些方法被抽象出来，单独写成函数，放在tools里面了。

目前，代码只写到这里。