**Subject**: Outlining the contents of codifying\_final\_code.py and RowObject.py

**Date**: July 31, 2021

**From**: Ruairí OCearuil, Data Intern, Cook County SAO, [ocearuilru@gmail.com](mailto:ocearuilru@gmail.com)

**Background**

The purpose of this code is to take the Illinois Compiled Statutes (ILCS) in html format and turn them into a csv file that maintains the underlying ‘ancestral’ structure where paragraphs that are subsections are properly attributed to their ‘parents’ as well as any ‘children’ that subsection may have. The original project used only the 2020 law book, however unless there are significant changes to the html formatting of the law book, it should be generalizable (with a bit of tweaking) for any year.

The file named *Codifying\_ILCS.py* does a majority of the work, with *RowObject.py* holding a secondary python class named (cleverly) RowObject.

I have loosely broken the code into two parts: (1) Reading in the html file and constructing a list of RowObjects and (2) Turning the RowObject list into a csv where each row keeps crucial information from its ancestry. At times, the code has some very specific issue it is correcting that may seem trivial, but they were all issues we identified through the process of matching the table to the most frequent SAO cases and in many cases they were important cases that significantly increased matching rates.

Lastly, it is important to understand the underlying html of the law book in order to understand this code. In the 2020 version, every useful line of code was wrapped in a ‘p-tag.’ Once we grabbed all the p-tags, we had to identify the ‘class’ attributes that we needed. The most important was class=SECMAIN which held the heading information for a segment of text – Chapter, Act, Section, and Title. Then, the class=INDENT-# held the various sub-paragraphs. We determined that the deepest our book went was to INDENT-6, but this may change for other law books.

**The Code**

Below I will attempt to concisely explain what each function in the code is doing and I will also describe any actions that need to be taken on the part of the user (marked with a \*).

***Part 1: Read in html and construct a list of RowObjects***

**\*Step 0**: Update the *base\_path* variable to point to the correct folder on your desktop (which contains *wholebook\_2020.html* (or a new html if using a different law book)

**Step 1**: *read\_in\_data()* takes the html that [Step 0] points to and returns soup using BeautifulSoup’s built in xml parser.

**\*Step 2**: *tags\_of\_interest()* takes the soup from [Step 1] and returns only the p-tags with classes we care about (secmain, indents, history, source, and img). As the html for a project this big is rarely perfect, I had to hardcode a few lines of the html to correct their indentation. I did this if year == 2020 in the function. I have left space for the user to add in hardcodes to identify specific text and alter that rows indent value.

\***Step 3**: Helper Functions

* *parsable()* checks if a row is of the class Secmain and is, in fact, a RowObject
* *parse\_secmain\_text()* uses *parsable()* and then takes a Secmain row, which look like:
  + *§ 5 ILCS 100/1-5. Applicability.*
  + *Rule 434. Jury Selection*

And it separates this into Chapter/Act/Section/Title or Rule/Title.

The RowObject is then assigned these as attributes that it carries

* *inherit\_parent\_attributes()* takes any row object, finds the closest Secmain directly above it, and assigns that RowObject all of its ancestor’s attributes (wither Ch/Act/Sec/Title or Rule/Title)
* *check\_in\_text\_numerals()* uses a series of rules to determine of the text of a paragraph contains subsection that are not themselves separate paragraphs in html. It can check for subsections starting with (i), (1), or (A) and simply returns a keyword identify which of the three types the paragraph contains or just ‘None’.
* *identify\_text\_body()* takes the text of a row and returns an index of where the actual language starts rather than just the section marker (i.e “(7) The law…” would return 4.)
* *\*parse\_in\_text\_numerals()* uses *check\_in\_text\_numerals()* figure out what type of subsections a paragraph contains and splits the paragraph into those subsections.

I hardcoded the numerals for each of the three scenarios and only went out as far as I found the farthest subsection of each type in the 2020 book, this might have to be adjusted.

**Step 4:** *rowObjectify()* is the central function in Step 1, turning each of the html rows of interest into a RowObject that maintains its ‘heritage’ in a tree structure while also inheriting the necessary attributes of its ancestors to make each instance of RowObject a complete csv row. The function ultimately returns a list of RowObjects.

Going into a bit more detail, the function takes in all the html rows we’ve identified as important and checks several different conditions. I have broken the code into 6 cases.

* Case 1: If a row starts with just text (i.e. no subsection name like ‘(a)’ or ‘a.’) then it should be merged with it’s immediate ancestor’s text.
* Case 2: A few subsection headers which should read, for example, ‘(7.5)’ were inappropriately split with ‘(7.’ on one line and ‘5)’ starting the line that should actual start ‘(7.5).’ This if-suite corrects this.
* Case 3: This uses *check\_in\_text\_numerals()* and *parse\_in\_text\_numerals()* to see if there are subsections within that rows text and turn those subsections into their own RowObjects.
* Case 4: Deals with the case of rows where class = source which need to have some weird characters replaced. They walk backwards and give their text to all their ancestors without actually being added to the final list of RowObjects.
* Case 5: Deals with rows where class = history in a similar way to sources, since history rows are essentially source rows for specific sections of the law book
* Case 6: Deals with SECMAIN rows by stripping out all the useful attributes that will then be handed down to descendants. If the row is of the class SECMAIN, it is sent through the *parse\_secmain\_text().*

***Part 2: Use the master list of RowObjects to construct a CSV***

Here we are writing a csv outside of any function by looping through the master list of RowObjects and putting each of them through *RowObject\_2\_csv\_row()* to extract all the column information and put it into a dictionary that becomes a row in the csv. *RowObject\_2\_csv\_row()* treats SECMAIN rows differently because they don’t have text so it becomes a mostly blank row.