Introduction

Reformats. It’s a collection of scripts I wrote to facilitate and accelerate formatting text in Mobius. Each script has its own unique purpose, and in this document, I want to give an explanation for why each script is present so that any prospective users will be able to understand where I was coming from when writing these scripts.

To start off, what is scripting? According to [Wikipedia](https://en.wikipedia.org/wiki/Scripting_language),

A scripting or script language is a programming language for a special run-time environment that automates the execution of tasks; the tasks could alternatively be executed one-by-one by a human operator. Scripting languages are often interpreted (rather than compiled).

Ok, so we’ve got some programming mumbo-jumbo, some stuff about run-time environments and interpretation, but what’s important here is “a programming language… that automates the execution of tasks; the tasks could alternatively be executed one-by-one by a human operator”. That’s precisely it - the scripts presented in this folder exist to automate tasks that I, a human, didn’t want to do by hand.

Obviously, Mobius isn’t perfect. No platform is, and especially not one that aims to do something as complicated as educating the masses. In STEM, no less. A lot of it is really old, too, and built in a time and place where the sheer scope and eventual purpose of the platform was impossible to grasp. As such, it has some problems, many of which are repetitive, recurring issues that tended to get on my nerves. Things like spaces always appearing between coefficients and variables. Keyword definitions always needing to be formatted. Oftentimes, there’d be this one word or phrase that showed up all over the place, and it needed to be replaced by something else. These were all annoying issues that needed to be corrected, but I didn’t want to be the one to correct them.

Luckily for me, repetitive, recurring issues are something that computers are very good at dealing with. The Python programming language, especially, is equipped-and-then-some to deal with issues like these. Even better, it can access my clipboard! If only there was some way I could easily run my files, preferably with a single stroke of my hand and a couple customizable key presses. And that’s exactly the problem that AutoHotkey, a mini-scripting language itself, can solve for me. Now, just by pressing key combinations I myself created, I can run any script I like. Perfect.

AutoHotkey

If you’ve ever used [AutoHotkey](https://www.autohotkey.com/) before, you can skip this page, as I’ll be doing nothing more than going over how it works and what it’s doing when you press one of the key combinations that it’s been told to accept.

Let’s bring in an example to help illustrate. This is a snippet from Reformat.ahk.

PATH = C:\Users\che\Documents\Reformats

^Capslock::

Send, ^a

Send, ^c

RunWait, format.py, %PATH%

Send, ^v

return

AutoHotkey reads its language from top to bottom, so we can start looking at the top.

PATH = ... is a variable definition that’ll be useful in a sec.

The line “^Capslock::” basically tells AutoHotkey to, upon pressing Ctrl + Capslock (Ctrl is a special key with an abbreviation; the list of all abbreviations can be found [here](https://www.autohotkey.com/docs/Hotkeys.htm#Symbols)), run the code specified in the text below until reaching the return statement, or until the end of the file has been reached.

Send sends the input immediately following it to the computer (for the Ctrl + Capslock script, I’m sending Ctrl + a for Select All and Ctrl + c for Copy).

RunWait tells AutoHotkey to run the script specified immediately after it and wait for it to finish running (here it’s format.py) which is located in the folder specified by the PATH variable. What format.py does exactly is specified below.

After one more Send (this one is used for Ctrl + v, for Paste), the script finds the return statement, and AutoHotkey stops where its execution procedures.

Of course, AutoHotkey has many, many more commands and abilities (the full documentation can be found [here](https://www.autohotkey.com/docs/AutoHotkey.htm)) but these commands are what I found to be useful.

Scripts

Yeah yeah, I get it. You’re not here to listen to me ramble about how smart I am, you’re here to learn how you can harness this incredible power for yourself. Here’s a list of the problems I was able to identify in Mobius, and the [Python](https://www.python.org/) scripts I used to automate their solution. I’m assuming you know Python. Please tell me you do.

Problem: Equation Editor Default Formatting

In my honest opinion, Mobius’ Equation Editor is incredible. It rivals the equation entry intelligence of top editors like [GeoGebra](https://www.geogebra.org/?lang=en) and [Desmos](https://www.desmos.com/) in terms of immediate understandability, and by harnessing the full power of [MathML](https://www.w3.org/Math/) rendering, it’s capable of displaying basically anything that the last 2000 years of mathematicians have dreamt up. That said, it’s got some annoying habits that can’t be turned off, such as:

* When an opening or closing bracket/parenthesis/brace/absolute value bar (or any of the other paired operators) is typed without a partner, Mobius lets you know by highlighting the unclosed operator in #DAFFDA green. This is a good thing if that operator was left unclosed by accident, and it’s a good reminder that it has to be closed. This is a bad thing, however, if that operator was left unclosed on purpose, and the user knows that it’s partner will appear somewhere else eventually. Fixing this is done by removing the <mstyle incomplete='true' mathbackground='#DAFFDA'> tag from the MathML source.
* Whenever a number is followed immediately by a letter (i.e. a variable), a space is automatically inserted between the two. I’ve admittedly only been here for four months, but I have yet to find a situation where this is more useful than not. To get rid of the space, the corresponding <mo>&nbsp;</mo> needs to be removed from the MathML source of the equation. This can also happen with <mspace> tags.
* All letters typed in the Equation Editor are italicized by default. Collections of letters, also known as words, are then also italicized by default. However, we usually want our words to be straight, unformatted. This can be done by changing the “italic” in <mi mathvariant=”italic”>...</mi> to “normal”.
* Certain functions (i.e. trig, logs) should have <mo> tags surrounding them, rather than the <mi> tags that the Equation Editor gives them by default.

There are a couple other problems like this, but they all boil down to one or more MathML tags that need to be added, removed or altered from the default Equation Editor settings. While this can be done by hand, I wouldn’t advise it. Instead, let format.py do the work.

Solution: Line-by-line Reformatting

Here’s how format.py works. It starts by pulling all of the text off of the Clipboard (which, depending on the AutoHotkey script you’re running at the time, will be either all of the text in the MathML source or a specific portion that you’ve highlighted). It then splits the copied text into a series of lines, then each line (make sure each line only has one tag on it, otherwise the script won’t work properly) is parsed for certain details. You can think of the script as doing something like this:

for (every line) in (pasted text) do:

let toPaste = “” # an empty string

check if line needs to be edited

if line needs to be edited:

edit line

add line to toPaste

otherwise:

add line to toPaste # no editing

copy(toPaste) to Clipboard

To check if a line needs to be edited, certain conditions matching the problems discussed above are checked for, which can be viewed and modified in the script itself. In there, I’ve tried my best to comment and explain what’s going on, and it should hopefully look sensible enough to be understandable.

The script itself also maintains a list of modes, which are specific formatting possibilities that the script will look for. These modes can be added or removed from the list at the very top of the script whenever they need to be changed (often they do), so the exact performance of the script is customizable. Modes can also be added just by adding another ‘if’ statement, and can be removed by removing the if block from the code altogether.

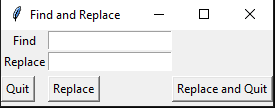
Problem: Recurring Word Replacements

Oftentimes, the text in a text page or question contains words that were used or phrased incorrectly. Less often but enough so to be a problem, those words are repeated throughout the text, as a false assumption or misunderstanding would be expected to cause. Heck, maybe there are too many rogue &nbsp;’s all over the place and they need to be gotten rid of. For that, I developed replaceAnywhere.py.

Solution: A Universal Find+Replace

The basis and code for this solution are very simple. Take all highlighted text (the particular hotkey combination for this script requires the user to highlight the text containing the words they want replaced first), copy it, replace all occurrences of one particular word with another particular word, repeat the last step if necessary, then paste the newly replaced text. Python itself even has a replace() method for strings, which does exactly the job of replacing all substrings in a string with another string. The only difficulty here is how the phrases to be found and replaced are obtained from the user.

I could have just used Python’s input() functions, but what if multiple words needed to be replaced? Then the script would have to be run multiple times. I decided that was boring, and instead opted for the Python module [tkinter](https://wiki.python.org/moin/TkInter), which allowed me to great a small GUI that looks like this:



(Actual size)

This window takes two words, entered in the Find and Replace cells, and replaces all instances of the word in ‘Find’ in the highlighted text with the word ‘Replace’ when the Replace button is clicked. If Quit is pressed, the script will paste whatever text it’s currently holding, and if Replace and Quit is pressed, both actions will occur.

If you know tkinter, feel free to modify the script however you like.

Problem: Adding Tags Is Slow

Not much to be said about this, except: What do I mean by tags? Just general markup language tags. There are a lot of cases where text just needs an extra tag surrounding it to make the formatting work out, but those tags are sometimes hard to remember and type, or even just annoying to have to type. That’s where addTags.py comes in.

Solution: Just Use Python

With Python, it’s but a single step to make a script that surrounds a piece of highlighted text with a tag. The exact tag is stored in a pair of dictionaries in the script itself, along with a mode number - change the mode number to change the tag. Adding tags should also be simple, as following the existing layout should work just fine. As of right now, I have options for smaller font, diagonal striking for cancellation and a few specific <span>’s for text.

But what about formatting text directly into MathML?

Problem: Using The Equation Editor Is Slow

Even with the power of the equation editor, Ctrl + Shift + E as an opening shortcut and a layer of scripts on top of that, having to use the equation editor is not optimal. In effect, all it does is surround text with the <math> … </math> tags, and add the necessary MathML formatting to the contained text. This is difficult, which is why the Equation Editor is so useful, but the additional functionality it provides can slow it down. Wouldn’t it be so much better if we could script existing text right into MathML immediately? Well, I couldn’t manage it. But maybe you can, and I’d commend starting by taking a look at Enmathenate.py.

Solution: Use Python To Mimic The Behaviour Of The Equation Editor

Of course, why didn’t I think of that earlier? The solution was right in front of me!

No. This is hard, which is why I wasn’t able to get the script to a level I’m proud of in the time I had. Right now, the script is able to handle basic numbers (with decimals, though accounting for that has its own issues), some Greek and English letters (more can be added) and basic elements (single elements and elements in covalent bond notation, i.e. H-H or Cl-Cl). More functionality is possible, and I hope it will get there.

Problem: I’m Lazy, But I Want My Text To Look Good

This is a huge problem. It’s such a big problem that I wrote all of the scripts in this folder full of scripts, hundreds of lines of scripts, just to deal with my own laziness. Most of the scripts I wrote are useful when dealing with genuinely annoying problems, but these next two were created purely out of my desire to do very small tasks very fast, because 10 seconds -> 1 second is still a 900% efficiency increase.

There are two problems in particular that I want to mention here, both of which deal with the ‘Algorithm’ box in Mobius questions. These are the problems of having spaces around math operators (+, -, =) in algorithm text, something I find is underdone despite it looking way better than having things be unspaced, and the problem of having to switch all +’s to -’s and vice versa (this comes up when, for example, authoring a question that is a negative variation of a different question). Two problems, two scripts, and those are operandSpace.py and opSwitch.py.

Solution: Use Python’s Capabilities For String Manipulation

Both of these problems have effectively the same solution. Just look through the text in the clipboard (AutoHotkey once again takes care of all of the copying/pasting, just highlight the text you want formatted and hit the hotkey) and replace certain combinations of characters to polish the formatting.

If any of +, - or = are detected without spaces surrounding them, they are replaced in the text with the spaces inserted.

If either of + or - are detected (and this is more difficult as it sounds, as we also have to look for &plus; and &minus;, as well as two different kinds of - for some reason), then they are replaced with their additive inverse counterpart.

These two scripts are pretty much set in stone, they do exactly what I wanted them to do and not really any more. If you’d like any other formatting style similar to the ones presented here, it should be reasonably easy to make a script similar to the ones I’ve made here, and design them to format text in exactly the way YOU want. Have it your way.

Problem: The Textbook Import Script Is Imperfect

Some background: Both [OpenStax](https://openstax.org/), a free, online, open-source textbook publisher, and Mobius use [XML](https://en.wikipedia.org/wiki/XML) to format their textbooks. Because of this similarity, many of Mobius’ initial textbooks are based on OpenStax textbooks, as they can be imported into Mobius relatively easy. This isn’t a one-to-one import, however. Many changes have to be made to convert the precise details within the XML from OpenStax-style to Mobius-style, and because Mobius supports a greater degree of interactivity, it must apply additional formatting so as to enhance the student experience over a raw .pdf.

Unfortunately, the conversion script doesn’t apply always apply this additional formatting. Figures and tables aren’t always typed in correctly. Textbook-specific div items, such as OpenStax Chemistry’s Link for Learning and How Sciences Interconnect, or even textbook-nonspecific Examples, are unaccounted for. Ordered and unordered lists are thrown completely out of whack. Mobius-exclusive additions, such as hover-tooltips and figure/label links are implemented incorrectly or not at all, and, of course, the challenge of MathML is largely ignored, as mathematical expressions remain in plain text. Once again, I wasn’t able to solve all of these problems. I did, however, start up a couple of solutions, which lie waiting in formatOpenStax.py.

Solution A: Using The Style Guide As A Template For Label Creation

formatOpenStax.py might just be one Python file, but it’s currently designed to solve four different problems (though I won’t go over the fourth, as it’s small and you can figure out what it’s doing by looking at the script yourself). Problem A is the problem of labels. When I say “label” what I mean is any point in the text where a figure or table is referenced, which is usually done with a “Figure [number]” or “Table [number]” placed somewhere in the text. These labels also serve as media links, and clicking on them will automatically scroll the page to the location of the figure/table.

Actually, the conversion script does a pretty good job with these. All that needs to be changed are the text inside the link and the link’s specific formatting. The text inside the link will usually need to say “Figure [number]” or “Table [number]”, and, as I can’t really determine the number from the text itself, this still has to be done manually. The formatting I can do, but this boils down to a simple string addition around the sides of the link, and the specifics of that string are held inside the replaceLinks() function inside the script. Just change the lS and lE variables, which are locally defined at the top of the function and hold the styling tags.

Solution B: Using The Style Guide As A Template For Tooltip Creation

Problem B is the problem of hover-tooltips, which are a set of <span> tags that can be placed around text in a Mobius textbook to allow readers to see a definition of a keyword when they mouse over the text. The conversion script is able to mark these keywords down, but using a script, it is possible to fill them in automatically.

To do this, the script uses two variables, defined at the top of the program.

**formatted**: This variable holds a string representing the shell of a tooltip.  
**source:** This variable holds the location of a manifest file.

If you don’t know what a manifest file is, it’s an XML file that holds the data of a Mobius class, like I mentioned earlier. The script makes tooltips by identifying the keywords that need tooltips in the source text of a Mobius text item/question, then it looks for their definition in the manifest specified by the source variable.

How can it do that? Well, it’s actually surprisingly simple. In a manifest, all definitions are denoted with the <dt> … </dt> tag set. All I have to do is read the data from the manifest (which Python so graciously allows me to do), search it for the definition (which is the aforementioned <dt> pair with the keyword text in the middle), grab the corresponding definition itself, paste it into the formatted variable, and add THAT into the Mobius text/question text (which should hopefully be in the clipboard, if you’ve been using AutoHotkey correctly). For good measure, I also check with and without setting all letters to lowercase, and with or without the last character of the keyword (just in case there’s a rogue s making things plural). I even do some basic MathML formatting on the definition, as that’s a pain to do normally. By using a separate function to identify all locations where a tooltip is needed, I can completely automate the process of making tooltips.

NOTE: To use this function, you must also download the manifest of the class you’re trying to use it in, unzip it, and specify where it’s located to the script, otherwise it won’t be able to find the definitions. Also, if you’re working on a textbook other than OpenStax Chemistry, you’ll need to replace the formatted variable with one that matches whatever’s been specified for your textbook. Just a heads up.

Solution C: Using The Style Guide As A Template For Figure Creation

Problem C is pretty simple to understand. Here’s what the source code for a converted figure might look like:

<figure id="FIG ID">

<figcaption><span class="OpenStaxIS-fig-label">FIG NUM</span> CAPTION</figcaption><span data-alt=”ALTTEXT" data-type="media" id="fs-idm141095104"><img alt=”AlTTEXT” data-media-type="image" src="IMAGE SOURCE" /> </span></figure>

and here’s what it needs to look like:

<figure id="FIG ID">

<img alt="ALTTEXT" class="center" data-media-type="image/jpg" src="LINK" />

<figcaption><span class="OpenStaxChem-fig-label">Figure NUMBER</span>CAPTION</figcaption>

</figure>

To get the figure text from A to B, I COULD take the string, rearrange parts of it to make it look nice (you might notice that parts of it are in the wrong order), relabel all of the tags and attributes, and delete the parts we no longer care about… OR, I could instead hold B as a template inside my script, and just grab the bits of information from A that I care about and inject them right into B. Yeah, let’s do that.

That’s it, actually, that’s all it does. The image template that the script uses is held in the script’s template variable, and you can swap out the style of the figure just by changing that variable. Just make sure that whatever you swap in has the strings “ALTTEXT”, “LINK”, “ID” and “CAPTION” inside it, as the script will use those substrings to determine where it needs to slot in the information from the old figure.

In the actual script itself, all three of these solutions (and a fourth) are combined together to run together, each right after the other. It’s important to note that the hotkey that runs this script selects the entirety of the question text that you run it on, and reformats the entirety of the text, so the hotkey will automatically select and copy all of the contents.

Solutions E-Z: Unimplemented

There are still many more things an all-in-one conversion-fix script like this needs to do. Here’s a list of problems I wasn’t able to implement solutions for.

NOTE: Many of these problems were based on text from the only textbook I had a hand in authoring the text for, which is OpenStax Chemistry. Other textbooks may have their own problems that I am entirely unaware of.

* Tables: These, like the figures, aren’t formatted perfectly, but they definitely could be with only a little more work.
* Examples + Solutions: These are little div blocks that tend to appear in many different textbooks, but for which the conversion script doesn’t implement perfectly. Solutions also sometimes have some extra Javascript that reveals a hidden answer to an extra practice question, which presently has to be inserted by hand. It’s completely feasible to make a script for this, though.
* Other Special Divs: In OpenStax Chemistry, there are special div blocks for the different special sections of the OSChem textbook, including the Links to Learning, the Portraits of a Chemist, Chemistry in Everyday Life and How Sciences Interconnect. I didn’t have the time to make a script for the purposes of formatting these divs, but if work is ever resumed on Chemistry, making a script that does this would be a good idea.
* Lists: Ordered and unordered lists (ols and uls) are completely torn apart by the conversion script, and have to be reinserted manually. This is really annoying, and while it might be a bit more difficult to do a script for these particular formatting issues, it would ultimately save a lot of time.
* Section Headers: I know this is easy to do manually, but why doesn’t the script do it? It would be so easy!

… and there are probably way more than I haven’t even noticed. I hope that readers of this document will be able to use the information I’ve laid out here to do what I feel is the purpose of this document - speed up the formatting of their Mobius work, and use my work as an inspiration for putting themselves ahead. Think about what I’ve done, and how it could be extended to solve other problems. Thank you. CH, Out!