**Beginner Tickets Documentation Cleanup**

By: Tanor Missler, Dwight Hohnstein, and Jessica Junk

Our final project was to clean up the documentation of four beginner tickets in Sage. The tickets were #8790 - logic.py, #8793 - boolformula.py, #8794 - logictable.py and #8795 - logicparser.py. For #8790 we also expanded the doctest coverage as requested on the ticket. In this paper we will cover our process in cleaning up the documentation of each ticket as well as a basic summary of the overall logic section of Sage. Our main resources for cleaning up the documentation were the [Sage developer’s guide](http://www.sagemath.org/doc/developer/conventions.html) and analyzing the documentation of other sections in Sage that had been updated within the last year.

There were five main features common throughout the different tickets that needed to be changed. The first common change was the input and output block formats had to be altered to fit the current standards. The picture below is an example of the change we made to each block.

Before:

INPUT:

self -- the calling object.

t -- a 2-D array containing the table values

vo -- a list of the variables in the expression in order,

with each variable occurring only once.

OUTPUT:

Effectively returns an instance of this class.

After:

INPUT:

- ``self`` -- the calling object.

- ``t`` -- a 2-D array containing the table values

- ``vo`` -- a list of the variables in the expression in order,

with each variable occurring only once.

OUTPUT:

- Effectively returns an instance of this class.

As you can see each input now has a dash in front and the variable has two back-ticks on each side. There is an empty line added between the title line and the body of the block. The output body now has a dash in front of it. All body lines of the block are indented to four spaces. If there is a run on line in an input line the secondary line is aligned with the first back-tick of the first line. If there was a run on line in the output it was aligned with the first letter of the top line. These changes were made to all the input/output blocks in each ticket.

The second common change was the separation of examples from each other. In the previous format there was simply blank lines between samples but with the new format a “::” needed to be added in between each line. Another sample image is shown below.

EXAMPLES::

Example 1

::

Example 2

OR

EXAMPLES:

Example description

::

Example 1

As we can see above each example is separated by “::” and a blank line however, the examples title only has a “::” at the end if there is no example description. If there is an example description the “::” comes after it instead.

The third common change is that the format for each note type block had to change. These blocks include note blocks, warning blocks, see also blocks, and todo blocks. These blocks follow after the examples block or the error block if one exists. Below is an image of the general changes for the format of the blocks.

Before:

Note:

Note description

After:

.. Note::

Note Description

As we can see the new format includes two periods, a space, the block type, and then two colons. There is also now a blank line between the title and the note description. The note description is indented by four spaces.

The fourth common change was the format of the error blocks. Error blocks come directly after the examples block. The change was relatively simple and involved being placed in a common statement and ending with “::”. Below is an image of the general change made to the error blocks.

Before:

Errors:

Error examples and the throwing types

After:

It is an error to …::

Error examples and the throwing types

As we can see the new format is the statement “It is an error to” followed by a short description of the error then two colons. There is a blank line between the title line and the body line. If there are multiple errors simply

The final common change was in the descriptions of each function all the variables and fields had to be surrounded by double back-ticks. Classes had a different format as shown in the image below.

Before:

def get\_bit(x,c):

r”””

This function is for internal use by the class SymbolicLogic.

It returns bit c of the number x.

After:

def get\_bit(x, c):

r"""

This function is for internal use by :class:`SymbolicLogic`.

It returns bit ``c`` of the number ``x``.

As we can see the parameters are surrounded by double back-ticks and the class has “:class:” in front of it and the name is surrounded by single back-ticks.

These were the five common changes among the tickets. Other changes that were a little more specialized included spelling corrections, fixing indentations, adding in titles/descriptions, and in one case moving the documentation from a function into the class due to the function starting with an underscore. Functions starting with an underscore do not show up in the reference manual so in logictable.py the \_\_init\_\_ function documentation was moved out into the class. This also will allow the use of “TruthTable?” at the “sage:” prompt to show the docstring because previously it did not.

To get an idea of how we can use the logic section of Sage, we will now look at some of the functions we can call in logic.py, logicparser.py, and logictable.py. From logic.py we use the SymbolicLogic class to make a truth table. This can easily be done by using the statement, truthtable, and print\_table methods as shown in the example below.

sage: log = SymbolicLogic()

sage: s = log.statement("a&b|!(c|a)")

sage: t = log.truthtable(s)

sage: log.print\_table(t)

a | b | c | value |

---------------------------------------

False | False | False | True |

False | False | True | False |

False | True | False | True |

False | True | True | False |

True | False | False | False |

True | False | True | False |

True | True | False | True |

True | True | True | True |

As seen in this example, the statement method creates a list to be turned into a truth table, by the truthtable function. The print\_table method formats the truth table to print to the terminal window. There is one other useful method in the SymbolicLogic class in logic.py, the method combine. This combines two statements into one new statement separated by the operator ``or``.

Now we will look at three methods in logicparser.py; parse, tokenize, and tree\_parse. The parse method takes a string ``s`` containing a boolean formula and returns the tuple of (parse tree of ``s``, variables in ``s``). The tokenize method is similar to the parse method, however it returns a tuple of (tokens in ``s``, variables in ``s``). Now we will look at the tree\_parse method.

sage: import sage.logic.logicparser as logicparser

sage: t = ['(', 'a', '|', 'b', '&', 'c', ')']

sage: logicparser.tree\_parse(t)

['|', 'a', ['&', 'b', 'c']]

A seen in the above example, tree\_parse takes in a list of tokens and returns a parse tree of those tokes. The logicparser.py module creates and modifies parse trees like this one.

Finally, we will look at logictable.py, which creates and prints truth tables. Here we will take a look at two functions in the TruthTable class; latex and get\_table\_list. These two methods are similar in that they both take a table made by the TruthTable class and return a representation of the truth table, but their output looks quite different.

sage: import sage.logic.propcalc as propcalc

sage: s = propcalc.formula("man->monkey&human")

sage: latex(s.truthtable())

\\\begin{tabular}{llll}human & monkey & man & value \\\hline False & False & False & True \\False & False & True & True \\False & True & False & True \\False & True & True & True \\True & False & False & False \\True & False & True & False \\True & True & False & False \\True & True & True & True\end{tabular}

sage: import sage.logic.propcalc as propcalc

sage: s = propcalc.formula("man->monkey&human")

sage: s.truthtable().get\_table\_list()

[['man', 'monkey', 'human'], [False, False, False, True], [False, False, True, True], [False, True, False, True], [False, True, True, True], [True, False, False, False], [True, False, True, False], [True, True, False, False], [True, True, True, True]]

As seen in the above examples, the latex function returns a LaTeX representation of the truth table. The get\_table\_list function returns a list representation of the table.

Beyond the basics of the current implementation of logic.py and simple documentation clean up, we also implemented the combine functionality in logic.py. The function combine takes two logical statements, such as “a&b” and “c|!d” and returns a statement with these two propositions combined using the “OR” operator. Combine can take a string, a statement object gotten from calling SymbolicLogic.statement(“statement”) or any combination of the two. If it receives improper inputs, the function will raise a TypeError. The docstring and doctest coverage of this function were implemented by following the guide lines laid out above.

This gives a general summary of what can be done in the logic section in Sage. The functions and the examples considered here give a synopsis of the modules logic.py, logicparser.py, and loctictable.py. In addition, our work has ‘beautified’ in a sense the actual code of these modules, and even in some case enhanced functionality, as in that of logic.py. We leave the implementation of simplify(self, table) and prove(self, statement) in logic.py to other developers in the community.