EECS 481 HW6b Report

1. **Names and email ids:** Krithika Malayarasan (Uniqname: krithikm, Email: krithikm@umich.edu), Coy Catrett (Uniqname: ccatrett Email: ccatrett@umich.edu)

2. **Selected project:**
   The project we choose is SymPy. Here is a github link for the project: https://github.com/sympy/sympy and the website's link: https://www.sympy.org/en/index.html. It is a python library for symbolic mathematics. SymPy is entirely written in Python and they try to keep their code as simple as possible in order to be comprehensible and easily extensible.Their long-term goal is to become a full-featured computer algebra system (CAS) that's capable of performing a variety of mathematical operations. It offers functionalities for basic arithmetic, algebra, calculus, combinatorics, psychics, and more. This makes it an ideal tool for people who are in this space and are interested in science, engineering, and mathematics. They are able to use it in both educational and professional applications. The decision to choose SymPy over other open-source projects was dependent on several factors. Firstly, we wanted to choose an open-source project that we believed had a good impact on a community of learners and professionals. And we think Sympy does this well since they believe in providing easily accessible educational tools for people in the STEM field to benefit from. Next, we also noticed that this project had an active community which we thought would be a welcoming environment for new contributions since this is our first time contributing to an open-source project. Lastly, we also wanted to choose a project which used languages that we were mostly familiar with and since we both have had experience with Python in the past, we thought this project would be a good fit for the both of us.

3. **Social Good indication:**
   No, this project does not contribute to Social Good.

4. **Project context:**
   Business Model and Motivation
   Sympy was started in 2005 by lead developers, Ondřej Čertík and Aaron Meurer. And it has grown since as an open-source project with the help of many contributors. Sympy operates under the Berkeley Software Distribution (BSD) license, promoting both commercial and non-commercial use. The motivation to build this project is to have a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. It is a library that can be embedded in other applications and extended with custom functions as well as used as an interactive tool. It is a free and open tool for mathematics that anyone can use, modify, and extend due to it being open source.

Competing Projects
Sympy competes with both open source and closed source computer algebra systems. Here are a few competitors listed below.

- Mathematica: Wolfram Mathematica is a software system with built-in libraries for several areas of technical computing that allow machine learning, statistics, symbolic computation, data manipulation, etc. While this tool is highly powerful, it is costly, which can be a barrier for students and researchers.
- Maple: Maple is a symbolic and numeric computing system as well as a multi-paradigm programming language. Like Mathematica, the cost of this tool could prevent some users from being able to use it.
- Maxima: Maxima is a free tool for performing computer algebra calculations in mathematics and the physical sciences.
- Symbolic Math Toolbox (MATLAB): MATLAB provides tools for symbolic computation and algebraic computations. It is used widely by many users from different backgrounds such as engineering, science, and economics.

Overall Importance
Sympy is an important project because it helps provide a free, accessible tool for symbolic computation. This helps foster innovation and learning in mathematics, science, and engineering. It can be used by many different users such as students, educators, and researchers, etc. Sympy being python based also helps it integrate well with the scientific Python ecosystem, including libraries like NumPy, SciPy, and Matplotlib. Overall, Sympy exists as a free and open tool that supports a wide range of mathematical operations.

5. Project governance:

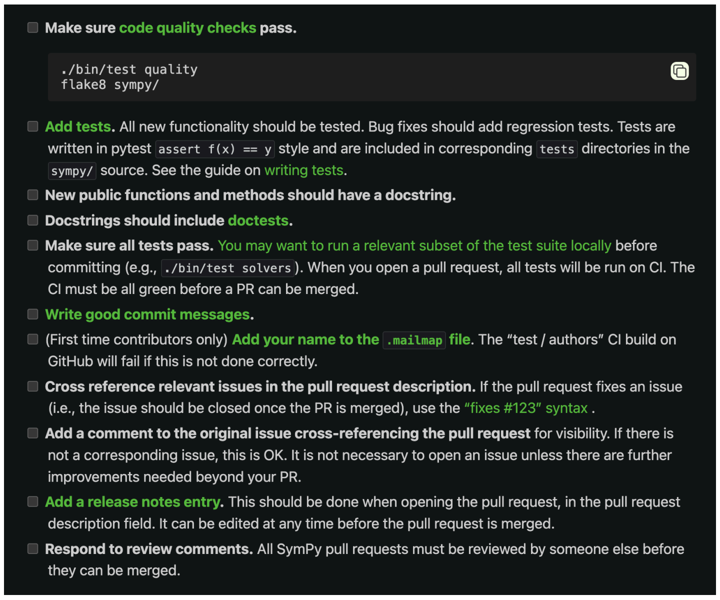
Sympy uses a collaborative environment where both formal and informal processes are used to engage its community of collaborators. This allows for an inclusive and productive environment for development, issue resolution, and feature enhancement.

Communication Tools
The main tool that Sympy uses is GitHub for its code repository and version control. GitHub is also the platform that Sympy uses for issue tracking, pull requests, and code reviews. For more immediate or informal discussion, contributors use Glitter and the google community group. Glitter is a chat and networking platform which allows for real-time communication and quick feedback. This can help with spontaneous discussions, immediate feedback, and troubleshooting. They also have a google community where people are able to ask any questions or discuss the SymPy project in any way. Additionally, SymPy maintains a mailing list, which is pivotal for more detailed or long-term discussions that aren't suited to the immediate nature of chat or the formal

structure of GitHub issues. The project also supports a wiki and comprehensive documentation hosted on GitHub, serving as vital resources for both new and established developers, guiding their contributions and offering detailed information about the project's operation and use.

Acceptance Process for Contributions

The acceptance process involves several stages to ensure the code quality and that it aligns with the overall project standards. First the issue is identified and a contributor creates an issue on Github detailing the bug, enhancement, or feature that they are proposing. They make sure to include details of the issue and any thoughts or ideas they have to start fixing the issue. Once this contributor posts the issue, other contributors are able to comment and continue the conversation about the issue. Then the contributor who wants to make changes to fix the issue makes a pull request (PR). They fork the repository, develop their feature or fix the issue in a separate branch, and they then submit a PR. The PR description describes what they did and it links back to the original issue they were trying to address. Next, after they submit their PR, automated tests are run to ensure that the new code doesn't break the existing functionality. Then, other community members and maintainers review the PR for code quality, adherence to coding standards, and overall effective integration with the rest of the project. They provide feedback and the contributor that made the PR is able to make revisions that they see fit based on this. Contributors are also expected to update or add documentation based on the new features or changes that they made in order to maintain the project's usability and accessibility for future users and contributors. Lastly, after approval by the maintainers, the PR is merged with the main code base to fully fix the issues and the issue is then closed. A screenshot is shown below on the checklist of things that need to be done when making a pull request to SymPy that Sympy includes in their documentation.



☐ **Make sure code quality checks pass.**

```
./bin/test quality
flake8 sympy/
```

☐ **Add tests.** All new functionality should be tested. Bug fixes should add regression tests. Tests are written in pytest `assert f(x) == y` style and are included in corresponding `tests` directories in the `sympy/` source. See the guide on writing tests.

☐ **New public functions and methods should have a docstring.**

☐ **Docstrings should include doctests.**

☐ **Make sure all tests pass.** You may want to run a relevant subset of the test suite locally before committing (e.g., `./bin/test solvers`). When you open a pull request, all tests will be run on CI. The CI must be all green before a PR can be merged.

☐ **Write good commit messages.**

☐ (First time contributors only) **Add your name to the `.mailmap` file.** The "test / authors" CI build on GitHub will fail if this is not done correctly.

☐ **Cross reference relevant issues in the pull request description.** If the pull request fixes an issue (i.e., the issue should be closed once the PR is merged), use the "fixes #123" syntax .

☐ **Add a comment to the original issue cross-referencing the pull request** for visibility. If there is not a corresponding issue, this is OK. It is not necessary to open an issue unless there are further improvements needed beyond your PR.

☐ **Add a release notes entry.** This should be done when opening the pull request, in the pull request description field. It can be edited at any time before the pull request is merged.

☐ **Respond to review comments.** All SymPy pull requests must be reviewed by someone else before they can be merged.

6. **Task description**

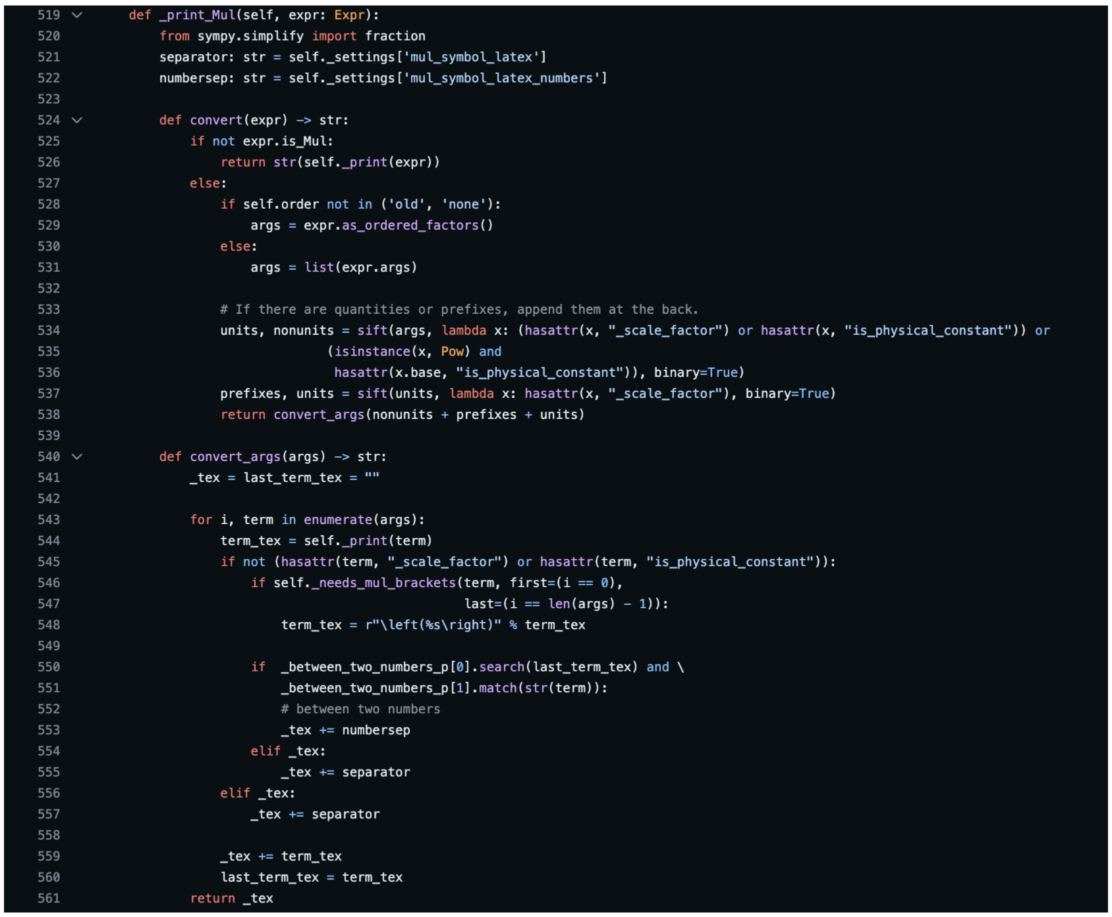
<u>Task #1 (Number 3 in HW6a Report): Extra \cdot in LaTeX output for Mul between a numeric power and a polynomial with numeric coefficient since SymPy 1.10 #26430</u>

Objective

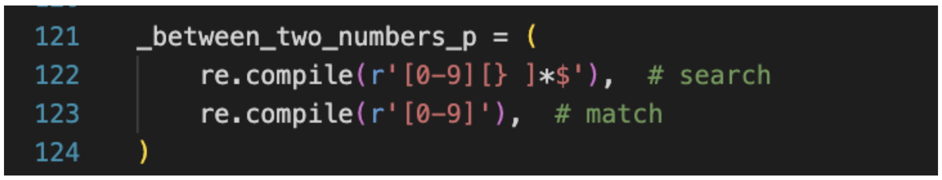
This issue involves an extra \cdot appearing in the LaTeX representation of the multiplication operations between a numeric power and a polynomial with numeric coefficient. This issue was introduced in the Sympy 1.10 version which is different from the output we were seeing with Sympy version 1.9. After some testing we noticed that the rendered LaTeX inappropriately inserts a '\cdot' symbol between a superscripted exponent and a polynomial wrapped within \left(...\right) brackets. This is specifically when the polynomial has a coefficient other than 1. The inappropriate insertion of the multiplication symbol could potentially lead to misinterpretation of the expressions.

Implementation

We first looked into which specific files were associated with this issue. Our approach for this issue was that we would first look into Mul and Pow operations within the LaTeX printing system to see why an extra cdot is being introduced here. And we noticed that it was specifically in the sympy/printing/latex.py file where the multiplication latex printer was defined. This is shown in the screenshot below. Specifically lines 550-553 are where the cdot is being added based on the _between_two_numbers_p function which is also shown in the screenshot below this.



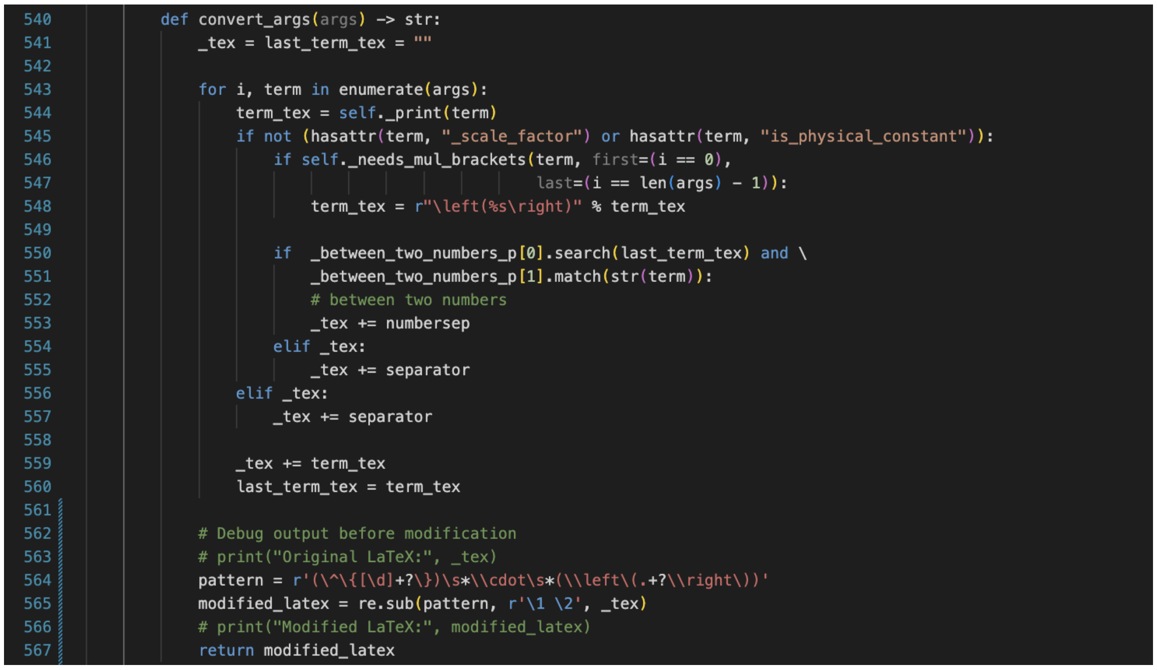
```
519     def _print_Mul(self, expr: Expr):
520         from sympy.simplify import fraction
521         separator: str = self._settings['mul_symbol_latex']
522         numbersep: str = self._settings['mul_symbol_latex_numbers']
523 
524         def convert(expr) -> str:
525             if not expr.is_Mul:
526                 return str(self._print(expr))
527             else:
528                 if self.order not in ('old', 'none'):
529                     args = expr.as_ordered_factors()
530                 else:
531                     args = list(expr.args)
532 
533                 # If there are quantities or prefixes, append them at the back.
534                 units, nonunits = sift(args, lambda x: (hasattr(x, "_scale_factor") or hasattr(x, "is_physical_constant")) or
535                               (isinstance(x, Pow) and
536                                hasattr(x.base, "is_physical_constant")), binary=True)
537                 prefixes, units = sift(units, lambda x: hasattr(x, "_scale_factor"), binary=True)
538                 return convert_args(nonunits + prefixes + units)
539 
540         def convert_args(args) -> str:
541             _tex = last_term_tex = ""
542 
543             for i, term in enumerate(args):
544                 term_tex = self._print(term)
545                 if not (hasattr(term, "_scale_factor") or hasattr(term, "is_physical_constant")):
546                     if self._needs_mul_brackets(term, first=(i == 0),
547                                                 last=(i == len(args) - 1)):
548                         term_tex = r"\left(%s\right)" % term_tex
549 
550                     if  _between_two_numbers_p[0].search(last_term_tex) and \
551                         _between_two_numbers_p[1].match(str(term)):
552                         # between two numbers
553                         _tex += numbersep
554                     elif _tex:
555                         _tex += separator
556                 elif _tex:
557                     _tex += separator
558 
559                 _tex += term_tex
560                 last_term_tex = term_tex
561             return _tex
```



```python
_between_two_numbers_p = (
    re.compile(r'[0-9][} ]*$'),  # search
    re.compile(r'[0-9]'),  # match
)
```

In the screenshot above, we can see the _between_two_numbers_p regex function. After reading some documentation about understanding regex, we were able to understand this function. The first regex pattern re.compile(r'[0-9][}]*\$*') appears to be used for searching. It matches a digit [0-9], indicating the end of a number followed by zero or more of either a closing curly bracket or space '[}]*' and ends with zero or more dollar signs \$*, which in LaTeX can signify the start or end of a math mode. The second regex pattern re.compile(r'[0-9]') is for matching and detects a single digit. These patterns are used to determine where to insert a multiplication symbol (like \cdot) between two numbers. So for the error, we were able to conclude that using these regex patterns might mistakenly interpret the end of the exponent and the start of the polynomial term as two separate numbers that require a multiplication symbol between them.

We decided to first look for a way to identify an exponent followed by a polynomial with a numeric coefficient, since this is where the above function was incorrectly adding a '/cdot'. We then realized that in LaTeX, this would always occur when we had a superscript exponent immediately followed by a \cdot and then a polynomial within \left(...\right). So we decided to use regex to identify this pattern in the LaTeX that was being returned by the covert_args function and replace each occurrence where the \cdot is found between these two groups with just a space. This would then maintain the connection between the exponent and the polynomial without the '\cdot' symbol. The code that we added to implement this change is shown below.



```python
def convert_args(args) -> str:
    _tex = last_term_tex = ""

    for i, term in enumerate(args):
        term_tex = self._print(term)
        if not (hasattr(term, "_scale_factor") or hasattr(term, "is_physical_constant")):
            if self._needs_mul_brackets(term, first=(i == 0),
                                        last=(i == len(args) - 1)):
                term_tex = r"\left(%s\right)" % term_tex

            if  _between_two_numbers_p[0].search(last_term_tex) and \
                _between_two_numbers_p[1].match(str(term)):
                # between two numbers
                _tex += numbersep
            elif _tex:
                _tex += separator
        elif _tex:
            _tex += separator

        _tex += term_tex
        last_term_tex = term_tex

    # Debug output before modification
    # print("Original LaTeX:", _tex)
    pattern = r'(\^\{[\d]+?\})\s*\\cdot\s*(\\left\(.+?\\right\))'
    modified_latex = re.sub(pattern, r'\1 \2', _tex)
    # print("Modified LaTeX:", modified_latex)
    return modified_latex
```

Though we weren't able to make changes to the code to fix it before it inserts the '\cdot' which would need to be done for future work to more effectively maintain the code, we were able to patch the issue for now and make changes so that it gives the correct output. Due to the complexity of the issue because of the various edge cases that would need to be considered, we decided this patch would be a more effective fix for now so that it covers many cases and fixes those scenarios rather than accidentally messing up the scenarios where it was originally working.

Task Description

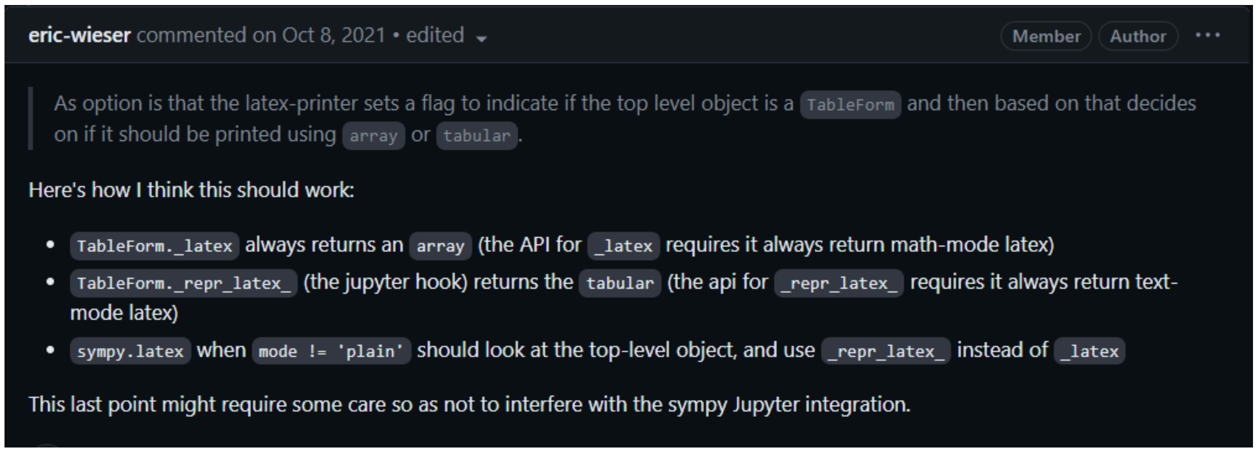
Task #2 (Number 2 in HW6a Report): sympy.TableForm produces invalid math-mode latex #20063

Objective

This issue involved printing apparently conflicting results when creating the latex for a table when using the tableform class defined in the SymPy library. When turning a SymPy tableform object into a table represented in latex by using the latex print method, there were some inputs whose outputs conflicted with the API of the latex printing method, i.e., printing a table using the tabular environment instead of with the array environment and using tabular inside of the mathmode environment.

Implementation

One possible fix suggested to us proposed that we change three parts of the source code to fix the issue. They are summed up in the following comment:



eric-wieser commented on Oct 8, 2021 • edited

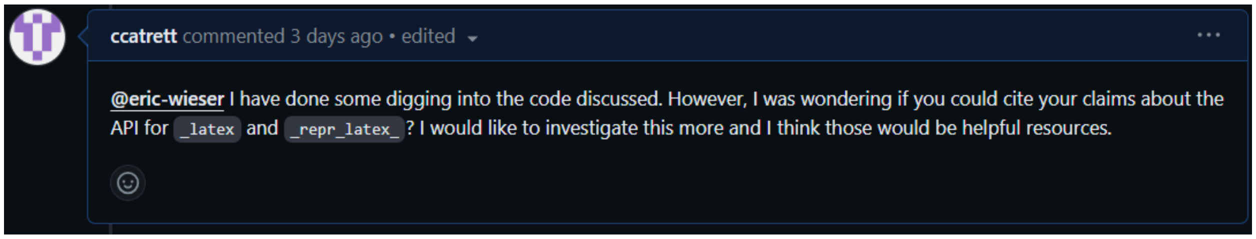
> As option is that the latex-printer sets a flag to indicate if the top level object is a `TableForm` and then based on that decides on if it should be printed using `array` or `tabular`.

Here's how I think this should work:

- `TableForm._latex` always returns an `array` (the API for `_latex` requires it always return math-mode latex)
- `TableForm._repr_latex_` (the jupyter hook) returns the `tabular` (the api for `_repr_latex_` requires it always return text-mode latex)
- `sympy.latex` when `mode != 'plain'` should look at the top-level object, and use `_repr_latex_` instead of `_latex`

This last point might require some care so as not to interfere with the sympy Jupyter integration.

We thought these changes sounded reasonable, but after some more research, we were unable to validate any of the API requirements after reading the documentation for SymPy and requesting citations of the claims made about the API of SymPy. We attempted to elicit more detailed requirements:


ccatrett commented 3 days ago • edited

@eric-wieser I have done some digging into the code discussed. However, I was wondering if you could cite your claims about the API for `_latex` and `_repr_latex_`? I would like to investigate this more and I think those would be helpful resources.

Further, we found strong conflicts with what this user was saying about the API and the documentation in SymPy available online: latex's print function defaults to plain tex mode, not math mode as eric-weiser claims. Without proper validation of the requirements demanded of us, we felt that it was unclear how to continue solving this bug. However, we still planned potential ways to fix this while waiting for a response from the above user. We read through the source code for this issue and thought about implementing checks within latex's print function which would use array environments instead of tabular environments when printing tables in math mode. This would at least solve the issue of printing tables with tabular inside of math mode, but may not have changed anything in the jupyter notebook hook.

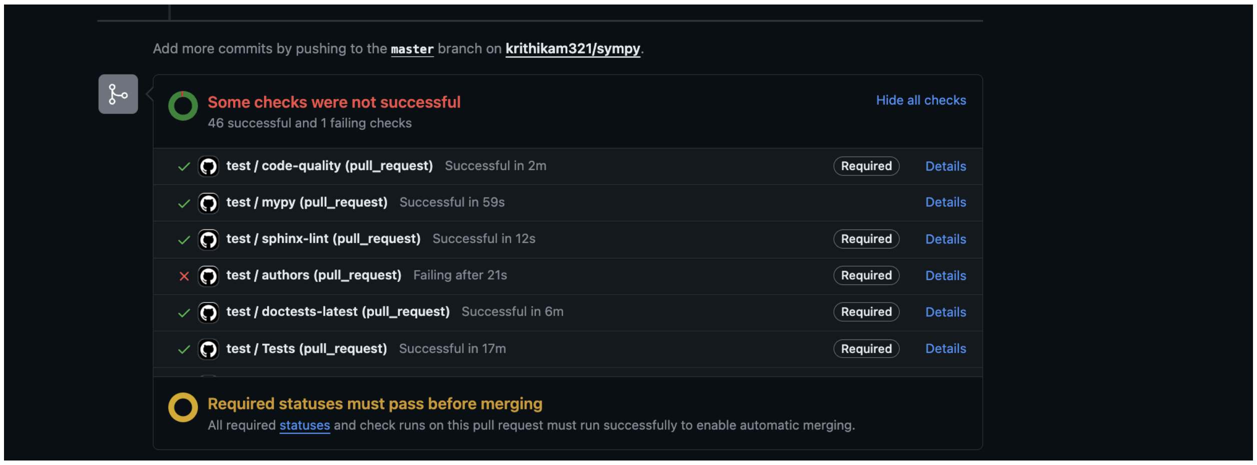
7. **Submitted artifacts**

Task #1 (Number 3 in HW6a Report): Extra \cdot in LaTeX output for Mul between a numeric power and a polynomial with numeric coefficient since SymPy 1.10 #26430

https://github.com/sympy/sympy/issues/26430
This is the link to the issue.

https://github.com/sympy/sympy/pull/26540
This link goes to the branch where I pushed my changes to fix this issue.

The screenshot below shows that the automated test cases that were run by SymPy when we pushed the changes did pass which ensures that the original functionality was preserved after making modifications. The only check that was failing was the test/authors (pull request) which is due to the fact that we are not verified authors for SymPy which is to expected since we would need verification from the maintainers. A screenshot of that is also shown below. But all tests about the functionality of the code did pass as shown below.


Add more commits by pushing to the `master` branch on krithikam321/sympy.

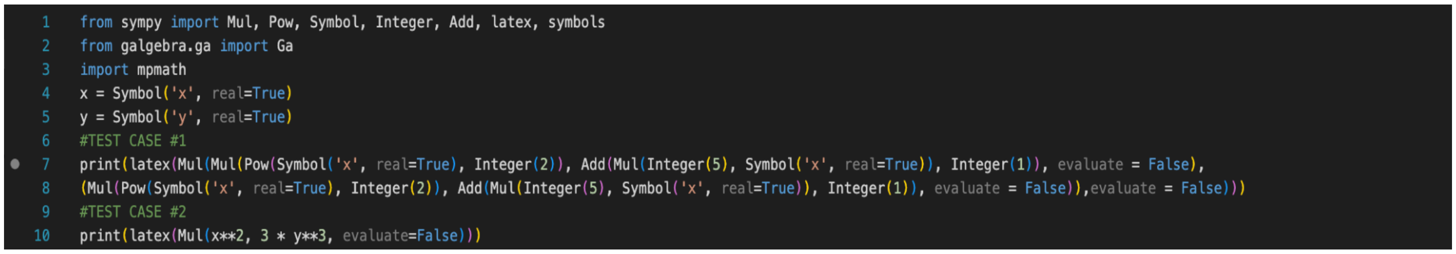
Some checks were not successful
46 successful and 1 failing checks

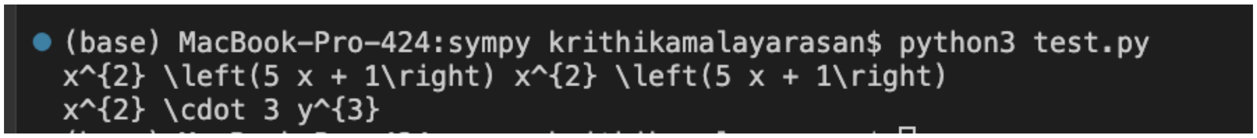
Hide all checks

✓	test / code-quality (pull_request) Successful in 2m	Required	Details
✓	test / mypy (pull_request) Successful in 59s		Details
✓	test / sphinx-lint (pull_request) Successful in 12s	Required	Details
✗	test / authors (pull_request) Failing after 21s	Required	Details
✓	test / doctests-latest (pull_request) Successful in 6m	Required	Details
✓	test / Tests (pull_request) Successful in 17m	Required	Details

Required statuses must pass before merging
All required statuses and check runs on this pull request must run successfully to enable automatic merging.

We created test cases of our own as well in order to make sure we were producing the correct output. We created more test cases to test the functionality as well as to understand where the issue started from. Shown below are two examples of test cases that we created with the output shown below which is what the correct output should be after the fixes. More QA activities that we performed will be shown in the next section.

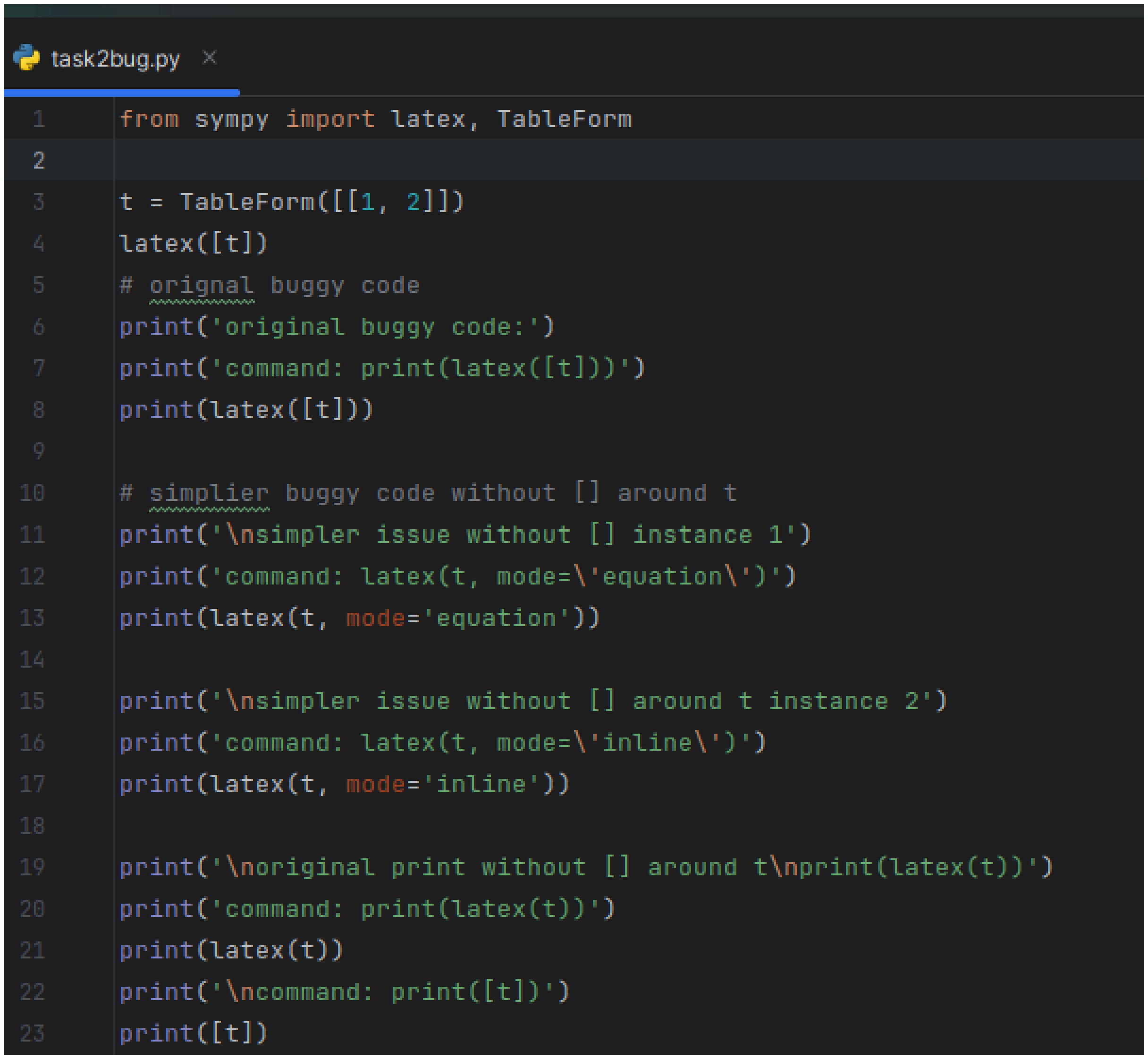

```python
from sympy import Mul, Pow, Symbol, Integer, Add, latex, symbols
from galgebra.ga import Ga
import mpmath
x = Symbol('x', real=True)
y = Symbol('y', real=True)
#TEST CASE #1
print(latex(Mul(Mul(Pow(Symbol('x', real=True), Integer(2)), Add(Mul(Integer(5), Symbol('x', real=True)), Integer(1)), evaluate = False),
(Mul(Pow(Symbol('x', real=True), Integer(2)), Add(Mul(Integer(5), Symbol('x', real=True)), Integer(1)), evaluate = False)),evaluate = False)))
#TEST CASE #2
print(latex(Mul(x**2, 3 * y**3, evaluate=False)))
```


```
● (base) MacBook-Pro-424:sympy krithikamalayarasan$ python3 test.py
  x^{2} \left(5 x + 1\right) x^{2} \left(5 x + 1\right)
  x^{2} \cdot 3 y^{3}
```

Task #2 (Number 2 in HW6a Report): sympy.TableForm produces invalid math-mode latex #20063

https://github.com/sympy/sympy/issues/20063
This is the link to the issue

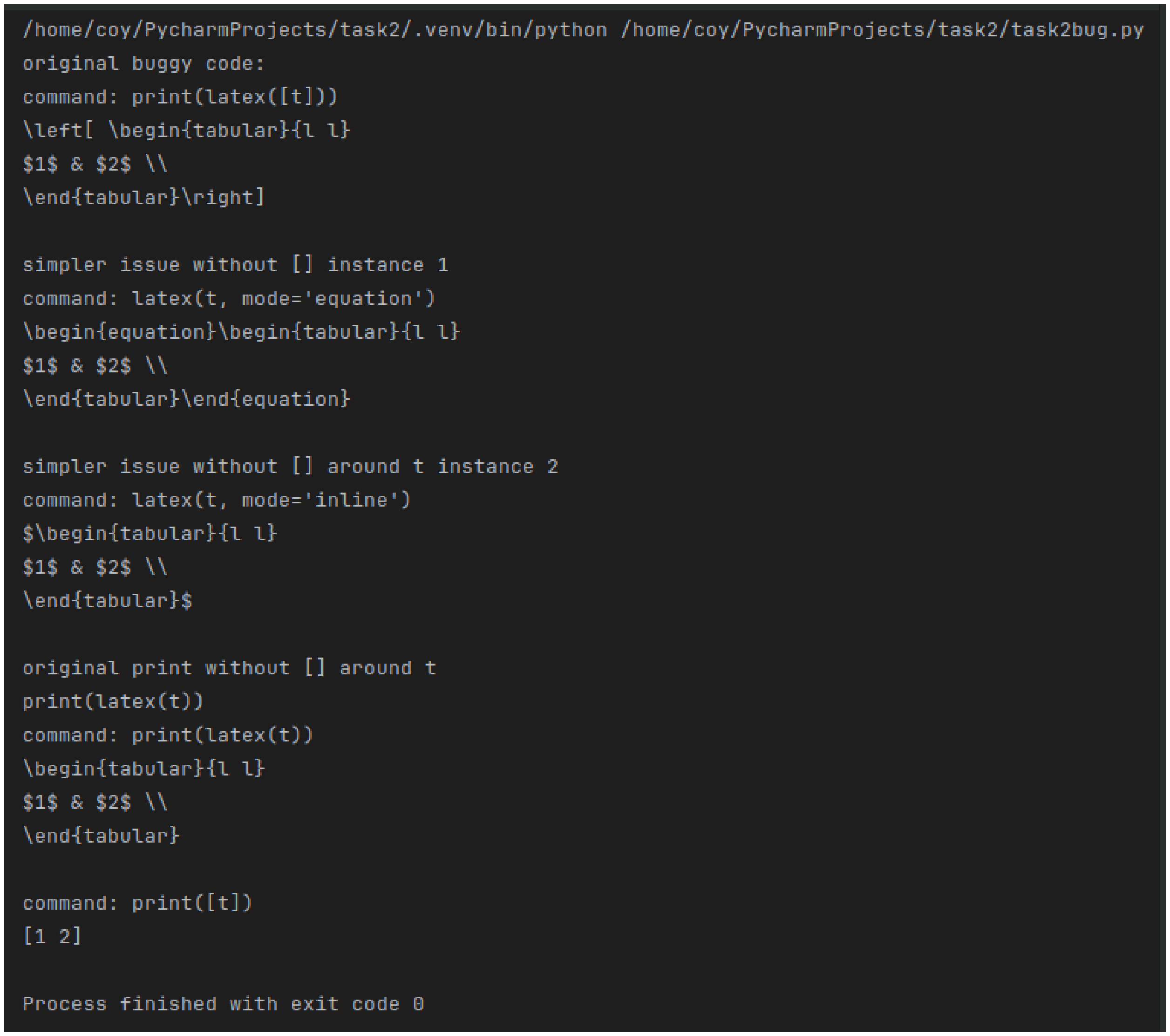
We created test cases to try to understand the behavior of the print.py and latex.py files in an attempt to make a possible fix for this bug; they are shown below.



task2bug.py

```python
from sympy import latex, TableForm

t = TableForm([[1, 2]])
latex([t])
# orignal buggy code
print('original buggy code:')
print('command: print(latex([t]))')
print(latex([t]))

# simplier buggy code without [] around t
print('\nsimpler issue without [] instance 1')
print('command: latex(t, mode=\'equation\')')
print(latex(t, mode='equation'))

print('\nsimpler issue without [] around t instance 2')
print('command: latex(t, mode=\'inline\')')
print(latex(t, mode='inline'))

print('\noriginal print without [] around t\nprint(latex(t))')
print('command: print(latex(t))')
print(latex(t))
print('\ncommand: print([t])')
print([t])
```

The output is down below as evidence of running it:



```
/home/coy/PycharmProjects/task2/.venv/bin/python /home/coy/PycharmProjects/task2/task2bug.py
original buggy code:
command: print(latex([t]))
\left[ \begin{tabular}{l l}
$1$ & $2$ \\
\end{tabular}\right]

simpler issue without [] instance 1
command: latex(t, mode='equation')
\begin{equation}\begin{tabular}{l l}
$1$ & $2$ \\
\end{tabular}\end{equation}

simpler issue without [] around t instance 2
command: latex(t, mode='inline')
$\begin{tabular}{l l}
$1$ & $2$ \\
\end{tabular}$

original print without [] around t
print(latex(t))
command: print(latex(t))
\begin{tabular}{l l}
$1$ & $2$ \\
\end{tabular}

command: print([t])
[1 2]

Process finished with exit code 0
```

Although the author of this bug, eric-weiser, stated that the tabular environment was not allowed inside of the math mode environment, the output of each of the test cases compiled in overleaf, an online latex editor. This allowed us to hypothesize that it wasn't an issue with SymPy and respecting the API for latex, but rather it was not respecting the API for SymPy. Again, we were unable to validate these hypotheses due to inadequate citations from the original thread and sympy's documentation.

8. QA Strategy

Our QA strategy was to test our fix on the minimal examples provided by the issue authors, as well as on more complicated examples that we contrived.

9. **QA Evidence**
   Evidence that our QA strategy was sufficient is reflected by the fact that our fix passed the repositories CI/CD protocol using GitHub Actions, and that our fix was pulled into the project repository.

10. **Plan Updates**
    We did deviate from our original plans. The main deviation was that we didn't anticipate spending quite as much time on just understanding the code, how everything was set up, and where exactly the issue was coming from by recreating the error on our own machines and seeing what other similar situations could cause the same error. A lot of our time was also spent reading SymPy documentation, LaTeX documentation, regex documentation, and python documentation to understand code and think about what changes we could make. Another deviation that we had was that we mostly planned to work on the tasks separately and check in with each other multiple times during the week. But we actually ended up mostly doing pair programming for the tasks due to the overall complexity of the task. So we would read documentation and come up with ideas before our meetings and then work on the code together during our meetings. We then ended the meetings with plans for what we wanted to do individually and when we would check back in to work on the code again. We ended up doing this due to the complexity of the tasks since working together helped us bounce ideas off each other and work on the issues more efficiently and effectively.

    Another main deviation that we had was the overall scope of the project. We originally picked out 4 tasks for both of us combined that we wanted to work on. But due to the complexity of the tasks as well as the unanticipated amount of time we would need to just understand everything to be able to think about ideas to fix the issue, we were only able to tackle 2 of the 4 issues. Some time was also spent waiting for responses from the contributors since we decided to leave comments on the issues and see if we could get any advice/feedback from contributors in order to better help us understand the issue. But since some of these issues were a little older, there was not much recent activity on them and thus not a lot of feedback was given to our comments which made it a little harder for us to start working on them since we lacked the background knowledge. We also looked at the google community that Sympy has to be able to gather any useful information from that as well. Because of all of this, we focused on fewer tasks but we believe this really helped us get more out of this experience and prioritize quality over quantity. By first comprehending the overall code, consulting with contributors, reproducing the issue on our machines, and iteratively developing and testing potential fixes across a range of edge cases, we achieved the necessary thoroughness for each task. Shown below in the table is an approximation of the time we spent on this project each week and what types of things we did each week to ensure the success of this project. We spent about a total of 30 hours on this project overall.

Week #	**Hours Spent**	**Activities performed this week**
Week #1	About 10 hours	Acclimating ourselves with the SymPy repo and the files in it, reaching out to contributors and joining the google community group, choosing the files where the error in our issues were coming from, learning more python syntax to better understand the code, reading SymPy documentation, and understanding LaTeX to better understand the issues and what the correct outputs should be
Week #2	About 12 hours	Waiting on responses from contributors, recreating the issues on our own machines in order to see what the correct outputs should be, making fixes for the tasks that we chose, continuing to read documentation to understand the latex python and regex, and starting the report
Week #3	About 8 hours	QA activities, making any other necessary changes to the code, and finishing up the report

11. **Experiences and recommendations**
Overall, our experiences with this homework were challenging but very rewarding. Both of us hadn't worked on an open source project in the past, so this was a very new experience for the both of us but because of this we felt like we were really able to learn a lot about working on such a large project. It was super interesting to work on a project with a lot of working parts and to understand how exactly open source projects work in general. Most of the projects that we work on in school are projects that we start from scratch and continue to the end. And they usually have concepts and a programming language that we learn right before the project, essentially equipping us with all the necessary skills to ensure success in the project. But this project was very different from that since this is a pre-existing, large-scale project so we needed to decipher all the working parts and identify and learn all the skills we needed on our own to be able to make effective changes to fix the issues. So this experience really was invaluable, teaching us to read code, implement fixes, debug, and do QA activities, all of which will definitely help us in our future software engineering roles.

12. **Advice for future students.**
You can let future students see our materials.

Krithika's Advice: My advice to future students taking EECS 481 is to attend all lectures, keep up with readings, and start on projects early to really get a lot out of the class and prepare yourself for working in industry as a software engineer.

Coy's Advice: Start homework six as early as possible. It will be a much more enjoyable experience for everyone involved and you will get more out of it. Additionally, the structured activities are good resources for clearing up some potentially confusing or unclear topics discussed in lecture.