CS130 Project Review
==========

Team performing review: bobcat
Work being reviewed: leopard

The first two sections are for reviewing the `sheets` library code itself, excluding the test code and other aspects of the project. The remaining sections are for those other supporting parts of the project.

Feedback comments on design aspects of the `sheets` library

Consider the overall design and structure of the `sheets` library from the perspective of the GRASP principles (Lecture 20) - in particular the principles of high cohesion and low coupling. What areas of the project codebase are structured in a highly effective way? What areas of the codebase could be restructured to have higher cohesion and/or lower coupling? Give specific suggestions for how to achieve this in the code.

Effective parts
- Creating a class for each of the functions makes it easier to associate more information with the functions which is good for maintainability.
- Decomposing and splitting each of the classes into its own type such as cell class, cellerror class, cellerrortype class, functions class and etc.

Improvements
- Dependency graph data is in the Cell objects and not just in the Graph class. The cohesion could be improved by keeping track of this information in just one place or the other.
- Graph class is high coupled with sheets/spreadsheet engine (not very generic). Graph.py could be more generalized (ex: rename_sheet, add_sheet, has_sheet don't have anything to do with a generic graph)
- Is there a reason you need convert_to_bool and convert_to_string in Functions.py when you already have convert_to in utils.py? This reduces the cohesion since there is a mix of the same checking being done.
- You could use the '?' operator to combine the four regexes for cell refs in your grammar. The corresponding lark code could be combined in Parser.py for these four regexes to provide better cohesion and reduce repetition.

Feedback comments on implementation aspects of the `sheets` library
---

Consider the actual implementation of the project from the perspectives of coding style (naming, commenting, code formatting, decomposition into functions, etc.), and idiomatic use of the Python language and language features. What practices are used effectively in the codebase to make for concise, readable and maintainable code? What practices could or should be incorporated to improve the quality, expressiveness, readability and maintainability of the code?

Effective parts
- Using a heap to keep track of the sheet extent and pushing/popping from heap
- Using copilot to generate code/tests 👍

Improvement
- Probably not necessary to have 3 versions of Tarjan's algorithm (tarjansCopilot, tarjansREC, tarjans in Graph.py), could use different branches to test these
- Some of the function names could be better such as the ones in workbook.py of add_children_cells and delete_children_cells.
- There is a lot of commented out code or code marked 'deprecated', perhaps could delete them when no longer needing it... git will track the history.
- There are a lot of helpful comments/docstrings for most of the functions (but there are also a lot of TODOs, can't tell if these were fixed already or not, like "TODO: fix this disgusting thing" in Parser.py)

Feedback comments on testing aspects of the project

Consider the testing aspects of the project, from the perspective of "testing best practices" (Lectures 4-6): completeness/thoroughness of testing, automation of testing, focus on testing the "most valuable" functionality vs. "trivial code," following the Arrange-Act-Assert pattern in individual tests, etc. What testing practices are employed effectively in the project? What testing practices should be incorporated to improve the quality-assurance aspects of the project?

Effective parts
- Organization of test files is neat, since they're all grouped by their project
- The tests seem pretty thorough and you test a lot of possible edge cases (like test_sort_cells_with_leading_spaces and test_sort_strings_resembling_numbers in test_sort.py)

Improvements

- There should be more integration tests that potentially exercises multiple different features so that more of the edge cases could be caught (like in test_sort.py, perhaps there could be more functions exercised before sorting)
- The performance test could be pulled out from the different suite tests when testing the correctness functions to really see the performance (could potentially add a timer to see how long it takes for a specific part of the function to test)

Consider the implementation quality of the testing code itself, in the same areas described in the previous section. What practices are used effectively in the testing code to make it concise, readable and maintainable? What practices could or should be incorporated to improve the quality of the testing code?

Generally, the code is pretty concise, readable and maintainable.

Effective parts
- Using pytest.fixture for a workbook for each test file is nice and reduces repetitive code
- Long test functions have good comments for what each block of code is testing (like test_acceptance.py)
- Most of the test files have comments that number each test and describe what it does, which is helpful (test_bruh isn't obvious though)
- The tests had the assert lines for checking whether a value is correct or not and checking the type when an error occurs

Improvements
- Some of the test file names could be named better
- Not really sure why the test_donnie.py is called that, but it's funny
- project_1_test.py is just commented out and test_pascals.py.old isn't being used either

Feedback comments on other aspects of the project

If you have any other comments - compliments or suggestions for improvement - that aren't covered by previous sections, please include them here.

Effective parts
- Having all of the helper functions in utils.py is good
- The README.md is helpful
- The cell error messages made me laugh

Improvements

- Perhaps the error messages could be a bit more descriptive of what error is occuring
- Perhaps the variable names could have better names (ex: in Parser.py, in the "function" function, the function is called function, there's a variable called function, and another variable called func, so this is hard to read)
- Big fan of the silly commit messages but I'm not sure Donnie feels the same