Report for assignment 3

Elin Fransholm (fransho),
Linda Nycander (lindanyc),
Mert Demirsü (demirsu),
Philip Ågren Jahnsson (philipaj),
Vilhelm Prytz (vprytz)

21 February 2025

1 Project

Name: click

URL: `https://github.com/pallets/click`

Click is a package for creating command-line interfaces. It offers an easy way to handle argument and options parsing and automatic help message generation. It is written in *Python*.

It has a branch coverage of about 85% and about 12000 NCLOC.

We forked this repo into `https://github.com/dd2480-2025-group7/click`. The code for the HomebrewCoverage can be found in a branch called coverage. The additional tests that improve the coverage is found in the forked repo.

2 Onboarding experience

It was sufficient to run *pip install click* to build the software, as documented. When building, "Coloroma" was installed too on Windows, and it concluded successfully. If you do not have pip you have to install it too, pip is very well documented.

To run tests, "Pytest" must be installed. For one computer who ran the tests on WSL (Windows Subsystem for Linux) it was sufficient to run *pytest* since Pytest was already installed. When running the tests, 627-629 pass, 1 fail and 21-23 is skipped, depending on the system/version.

We stuck with this project since it fulfilled all the requirements.

We plan to continue with this project.

3 Complexity

- `MissingParameter.format_message` in *exceptions.py* got CCN 13, both by calculating manually and using Lizard. The function is fairly complex, when analyzing CCN. The function is not very long, only 38 rows, including empty lines and comments, and 32 NLOC. The purpose of the function is to format the error message given to the user when some parameter or option is missing. The function does not have any exceptions, only one return statement. The class which the function is a part of has documentation, but the function itself has not.

- `open_stream` in *_compat.py* got CCN 11 with manual calculation and 21 with Lizard. The function has LOC = 79, including empty lines and comments. The function does not have any documentation. The purpose of the function is to open files/streams safely. It supports standard input/output and atomic writes. For atomic writing, a temporary file is created and written to first and afterwards this file replaces the original file to prevent corruption of files.

- `Parameter.__init__` got $CCN = 12$ with manual calculation (see this for reference on how it has been calculated manually), and 20 with Lizard. The function has $NLOC = 120$, which isn't too long. It is quite well documented using Python Docstrings which then automatically shows up on the API documentation. Exceptions are taken into account when we count the CCN manually. The usage of the `Parameter.__init__` function is to initialize an instance of the `Parameter` object. This object is used to indicate a parameter to a command in Click, such as `--example "bla"`. The documentation is somewhat clear with regards to the possible outcomes, since it is implied that the the `__init__` method of a given object creates an instance of this object. The documentation isn't really clear in regards to all possible outcomes as can be seen here, though this may be due to the fact that all exits are when an error is raised other than an implicit exit. The method is simply a constructor and thus does not return anything.

- `Option.__init__` in *core.py* got $CCN = 39$ by manual calculations (as can seen here), with Lizard counting it as 40. The reason for this difference is in our opinion most likely that the group counted implicit returns as an exit point which Lizard might not have done, this would reduce the functions complexity by exactly one. The function has $NLOC = 100$, which isn't too long. All input parameters are fairly well documented using the Python Docstrings, some variables used for error handling are inherited from the `Parameter` class and ass such do not get any explanation in the code as to what they represent. The usage of the `Option.__init__` function is to initialize an instance of the Option object which is used change certain options for the user.

- `HelpFormatter.write_dl` in *formatting.py* got a manual CCN of 6 and a

reported CCN of 7 by Lizard. The function spans 38 NLOC, contains 220 tokens, accepts 4 parameters, and covers 43 lines in total. Its purpose is to write a definition list—typically used for formatting options and commands—into the internal buffer of the help formatter. To achieve this, the code first computes the optimal column widths using `measure_table` and then determines the width of the first column. The control flow includes an outer loop ("for first, second in iter_rows(rows, len(widths)):") that contributes one predicate point and an inner loop ("for line in lines[1:]:") that adds another. In addition, three if-statements add further predicate points: one that checks if the second column is empty ("if not second:"), one that verifies whether the length of the first column fits within the allotted width ("if term_len(first) ¡= first_col - col_spacing:"), and one that determines if there are any wrapped lines ("if lines:"). The slight discrepancy with Lizard's CCN likely arises from an extra implicit decision point. Overall, the function is well-documented via a comprehensive docstring and clear comments.

The reason we could not choose the most complex and long functions is that those functions were already tested thoroughly, so we could not improve the branch coverage for them.

Functions with higher cc are not necessarily longer. There is some correlation but it depends from function to function.

4 Refactoring

- `Parameter.__init__` is possible to refactor for lower CC, but it may not be the right thing to do. Firstly, we could break down the validation logic in multiple "internal" functions (functions with a _ in the name, only to be used by `__init__`. Related validation checks could then be grouped together, such as `parameters`, `nargs`, `deprecated`. The downside is that we get a lot more methods, and slightly more lines of code. We also need to pass the state between validations methods. This can be tedious when debugging if logic is split into several different functions.
- `Option.__init__` has a couple of places in which it could be refactored to reduce cyclomatic complexity. However seeing as doing this would dramatically increase the amount of functions, it may not be a good decision. The complexity could be reduced by rewriting all decision points which does not have a return statement as subfunctions, which would include all decision points except for the error handling. However, by doing this to the `__init__` function, the readability of the code is drastically reduced as well. By changing all conditional statements between line 2541 and 2624, you could in theory reduce the CCN from 39 to 14. However as stated above, the readability of the code would probably be much worse, and tests would still have to be created for every additional function.

- `open_stream()` is possible to refactor. Some of the if-statements could be rewritten as subfunctions instead, which would reduce the complexity. For example the if-statements on lines 423 and 444. The code on line 444 sets file permissions of tmp_filename to match the original file's permissions. This can be moved to a subfunction reducing the cc by 1 since it removes a decision point. This can also be done for the if-statement on line 423 which changes the variable flag. The rest of the if-statements have exit-points inside of them and the cc would not be decreased if these were made into subfunctions. However the if-statements that are possible to move are only two lines each and would not decrease the cc a lot.
- `format_message()` is possible to refactor by dividing the function into several subfunctions. For example, the function has several if-statements that checks if the value of a class variable matches some string, and if they match, they set a variable to a certain string (a part of the error message that are being formatted). Therefore, those if-statements could easily be moved to another function, which simply returns the resulting string of the variable. This would decrease the CCN since we are moving decision-points (if-statements) to another function, and keeping all return-points within the function. However, the function is pretty simple as it is and therefore there is no need for refactoring it.
- `__getattr__` in *__init__.py* could be refactored to reduce repetition in handling deprecated attributes. One approach would be to use a mapping (such as a dictionary) that associates attribute names with a tuple containing the corresponding import function (or value) and its deprecation message. This would allow a single, unified code path to issue the deprecation warning and return the appropriate object, thereby reducing the CCN. However, while this refactoring would decrease redundancy, it might also obscure the explicit handling of each deprecated attribute and add an extra layer of abstraction. Given the function's already low complexity and its clear, self-contained logic, the benefits of such a refactoring serves no clear purpose.

5 Coverage

5.1 Tools

To get the coverage tool you run `pip install coverage`. It was easy to create a browsable html-file that the results of the coverage if you first run `python -m coverage run -m pytest` while in the project directory and then run python -m coverage html to make an interactive html file which shows code coverage for all functions. It was very easy to integrate with our build environment.

5.2 Your own coverage tool

We made a class called *HomebrewCoverage* for measuring branch coverage. For each function where you want to measure the branch coverage, you create an instance of this file and run the function *taken(i)* every time a branch is taken. The result from all runs will be written to a file.

5.3 Evaluation

Our tool measures the number of branches being tested and gives the result in percent (number of tested branches out of total branches). The tool we used gave the branch coverage in percent of statements (number of tested statements out of total statements). This means that our tool is not taking into account how many lines of code one branch is, even if it is more important to test a branch including many lines of code compared to a branch only including one line of code. Therefore, our tool does not get the same result as the tool from *coverage.py*.

6 Coverage improvement

- MissingParameter.format_message old: 55.6% new: 88.8%
- open_stream old: 19.2% new: 42.3%
- Parameter.__init__ old: 79.2% new: 95.8%
- Option.__init__ old: 83.3% new: 88.1%
- __getattr__ old: 50% new: 62.5%

We have added at least two tests for each function, some of our functions have 2 tests and some have a few more. These are the test cases we have added:

- For `format_message` we added 2 tests, which are named `test_format_message_missing_parameter` and `test_format_message_without_param_type` which can be found in file *test_options.py*. They are testing two cases where param_type is set to parameter and not set at all, since other param_types were tested but not these two. These tests increased the branch coverage for the function from 68% to 91% according to the tool *coverage.py*.
- For `open_stream` we added 3 tests, which are named `test_open_stream_raises_a`, `test_open_stream_raises_x`, `test_open_stream_raises_r_and_rb`. They test 3 different ValueErrors for different values of mode when atomic is true. The tests are in a new file called *test_open_stream.py*. The branch coverage was 75% and is now 90%.
- For `Option.__init__` we added 2 tests, which are named `test_prompt_with_non_boolean` and `test_hide_input_boolean_flag` which can be found in the file *test_options.py*. For these tests, errors are expected to be raised under certain conditions. Specifically, the first tests should raise an TypeError if the parameter

`prompt` is `True`, and `is_flag` is `True` while `flag_value` is not a boolean. The second test expects an TypeError to be raised while `prompt != None`, `is_bool_flag` is `True`, and `hide_input` is `True`. Using the tool *coverage.py*, the functions coverage went from 92% to 95%.

- For `Parameter.__init__` we added 2 tests named `test_nargs_mismatch_with_tuple_type` and `test_default_type_error_raises`. They both test ValueErrors. When adding the tests, the branch coverage increased from 84% to 98% according to *coverage.py*.
- For `__getattr__` in *__init__.py* we added 2 tests, which are named `test_getattr_basecommand` and `test_getattr_unknown` and can be found in the file *test_init.py*. These tests verify that when a deprecated attribute (specifically, `BaseCommand`) is accessed, the function correctly returns a callable and emits a `DeprecationWarning`, and that accessing a non-existent attribute raises an `AttributeError`. With these tests, the branch coverage of the function increased from 37% to 52%, according to the tool *coverage.py*.

7 Self-assessment: Way of working

This week, our group welcomed a new member, so we are now 5 people instead of 4. This is great, and we can now divide the work even more, compared to when we were only 4. However, this has led us to not improve our "way of working", according to the Essence standard. We are at *In use* since we are still getting to know each other and trying to integrate our new member into our way of working. We are almost back at *In place* with our new group constellation. Also, some people in our group have been sick this week, making collaboration a bit trickier. But we decided to have meetings over Zoom instead of our regular meetings on campus.

8 Overall experience

The main thing we learned was how important it is with documentation. It makes a significant difference for someone who is trying to understand how to use the tool or contribute to the code. For example, a well-documented onboarding experience, in a README or in an HTML-file (like in this project), makes it much easier to understand how to use the tool and start looking at the code. Code comments also make a big difference, especially on more complex code snippets, making it faster for someone to understand the code and write new code in the project.