

DD2480 - Coverage

Eloi Dieme,
Hugo Malmberg,
Olivia Aronsson,
Lovisa Strange,
Yuta Ojima

February 2024

1 Project

Name: Scrapy

URL: <https://github.com/scrapy/scrapy>

Scrapy is a tool for extracting information from websites.

2 Onboarding experience

Building the project only requires a working installation of Python (3.8+) and to install some packages. The project directory doesn't have a `requirements.txt` file to install all the dependencies directly. The easiest solution is to install the scrapy package using `pip` (preferably in a virtual environment) and then to try to run the tests (with `pytest` for example) to install the additional development dependencies. This whole process isn't really explained in the Scrapy documentation, which is more targeted at end users of the package. We have identified 4 dev dependencies (excluding `pytest`): `twisted`, `testfixtures`, `sybil` and `pexpect`. The tests run after all these packages are installed. Apart from that, in traditional Python fashion, no additional tool is required. What's more, Python being an interpreted language, there is no build process. Regarding the tests, they run with `pytest`, but some of them are skipped and some fail (but without stopping the execution). Coverage can also be easily measured with the `coverage` Python package, but it takes 20 minutes to run all the tests. All in all, the Scrapy project is not completely straightforward to "build" but we got around that pretty quickly and it checks out with the requirements of the assignment so we plan on continuing with it.

Function	Manual count	Lizard count
<code>_link_allowed()</code>	7	14
<code>run()</code>	13	13
<code>_process_spider_output()</code>	15	19
<code>_get_inputs()</code>	14	16
<code>_get_serialized_fields()</code>	14	14

Table 1: Result of the manual count of the cyclomatic complexity number compared to the count made with Lizard for the five functions

3 Complexity

3.1 Result of count

The result of the count of the cyclomatic complexity number, done by hand by the two group members for each of the five observed functions are presented in Table 1. This was computed using the formula presented in the lectures

$$M = \pi - s + 2,$$

where π is the number of decision points in the program and s is the number of exit points. Both group members found the same result for the functions. Also, the count that was computed automatically using **Lizard** is also presented in Table 1. In two of the cases, Lizard found the same number as the manual count. In two other cases, there was a small difference between the counts, and in one case, a large difference was found.

It is likely that some of this difference comes from how try-except-blocks are counted automatically compared to manually. There might also be some hidden complexity that can be found when automatically counting the cyclomatic complexity number, that we struggled to find by hand.

3.2 Length of code

The functions vary in length. Three of the functions (`_get_inputs`, `run` and `_get_serialized_fields`) are of about length 30 NLOC, and they also have similar cyclomatic complexity numbers. However, the `_link_allowed()` function is only 21 NLOC long, which is very short considering its high cyclomatic complexity number (at least the one computed automatically). On the other hand, `process_spider_output` is very long at 73 NLOC, so even considering that it has a very high cyclomatic complexity, it is still significantly longer than the other functions we have looked at. For the studied functions, we conclude that they are quite long, but also that they does not neccissarly have to be that long to be complex.

3.3 Purpose of functions

The purpose of the functions looked at are:

- `_link_allowed()` - The purpose is to check if the provided link is a valid link for the extractor
- `run` - The function is the entry point for running commands directly on the CLI using Scrapy.
- `_process_spider_output`- The purpose of the function is to process the output of middlewares used by Scrapy to add features to the scraping process
- `_get_inputs` - The purpose of the function is to find form elements in HTML responses collected by the scraper.
- `_get_serialized_fields` - The function returns fields to export as an iterable of tuples.

3.4 Counting of exceptions

When counting by hand, we found exceptions hard to account for when finding the total number. For two of the functions, namely `_get_inputs` and `_process_spider_output`, there are `try-except` blocks in the code. In both these cases, those blocks are probably the reason for the difference in result between the count using Lizard and the manual count. If we counted these blocks differently, we would probably have had a result closer to the one we got when counting automatically.

3.5 Documentation of the functions

Most of the functions have no real documentation. Some information about them can be found in the base class of the function, but it is only a docstring. Other functions do not have any documentation at all. More specifically, it is hard to find any descriptions of the many branches/outcomes covered by these complex functions.

4 Coverage

4.1 Tools

We used the python package `coverage` to compute the test coverage of the project. Like most python packages, it can be installed with the `pip` utility very easily and then to compute the coverage with `pytest`, we just need to run the command `coverage run -m pytest`. Then when it is done, the command `coverage html` generates a full HTML coverage report on the machine. All in all, `coverage` was very easy to integrate with the build environment and it is well documented so it was very easy to use.

4.2 Your own coverage tool

The code for gathering coverage measurements can be viewed by:

```
git diff master complexity-testing
```

4.3 Evaluation

1. How detailed is your coverage measurement?
It is as detailed as possible, it covers if-statements, loops and try-exceptions. However, some list comprehensions are not covered.
2. What are the limitations of your own tool?
If the code were to be changed, new branches would need to be identified and the log function would need to be called from that branch.
3. Are the results of your tool consistent with existing coverage tools?
Yes, the complexity is very similar to the one calculated by lizard and the coverage is very similar to the one gotten by running the coverage package.

5 Coverage improvement

After measuring the coverage using various means, our goal was to improve this coverage by writing additional tests.

- The initial measured coverage of the project can be seen [\[here\]](#).
- The measured coverage after implementing all new tests can be seen [\[here\]](#).

5.1 `_get_serialized_fields` function

5.1.1 Initial coverage

For the `_get_serialized_fields` function, initially 2 branches are not covered by the test suite out of 14 branches in total, as determined by `coverage`. So the initial coverage is approximately 85%. The breakdown of the function produced by `coverage` can be seen on Fig. 1.

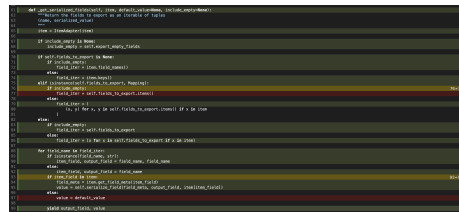


Figure 1: Initial coverage - `_get_serialized_fields`

5.1.2 Identified requirements

- **Req. 1** - Serialization of specified fields: the function serializes fields that are explicitly specified in the `fields_to_export` attribute. This includes handling both when `fields_to_export` is `None` (implying all fields should be serialized) and when it is a specific list or mapping of fields to be serialized.
- **Req. 2** - Handling of `include_empty` parameter: the function respects the `include_empty` parameter or attribute, determining whether fields not present in the item should be included in the serialization output with a default value.
- **Req. 3** - Default value for missing fields: when fields specified for serialization are not present in the item, the function can assign a default value to these fields in the output.
- **Req. 4** - Correct handling of field names and values in mappings: ensuring that when `fields_to_export` is a mapping, the function correctly maps item fields to their specified output names and serializes them accordingly. This includes verifying the correct handling of both present and absent fields, according to the `include_empty` flag.
- **Req. 5** - Behavior when `fields_to_export` is a mapping and `include_empty` is false: specifically testing the condition where fields are defined in a mapping (implying a rename is intended during serialization), and ensuring that only those present in the item are included when `include_empty` is `False`.

5.1.3 Untested requirements

Among the 5 identified requirements, 2 are untested as shown by the branch coverage on the function. These requirements are req. 4 and req. 5. So, we will write two tests, each targeting one of these requirements.

5.1.4 Test no. 1

The first added test is named `test_missing_field_yields_default_value` and is implemented in the `scrapy/tests/test_exporters.py` module. Its purpose is to test requirement 4. It does this by setting the `include_empty` flag to `true` and by trying to export data with a missing field. The assert statement is then:

```
assert serialized_fields == expected_fields, \
    "Should include missing fields with default values"
```

After this first test, the branch coverage has improved to 92% because one additional branch is now covered. The command to see the code using `git diff` (from the `improve-the-test-coverage` branch) is:

```
git diff 94aebd74e e37c8bc82 -- ./tests/test_exporters.py
```

The new coverage report can be seen on Fig. 2.

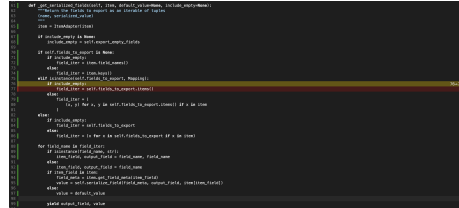


Figure 2: Coverage after Test no. 1 - `_get_serialized_fields`

5.1.5 Test no. 2

The second added test is named `test_mapping_with_include_empty_true` and is implemented in the `scrapy/tests/test_exporters.py` module. Its purpose is to test requirement 5. Specifically it tests that all specified fields in `fields_to_export` as a mapping are included when `include_empty` is `true`. It does this by setting the `include_empty` flag to `true` and by specifying a default value and then by trying to export data (as a mapping) with a missing field. The assert statement is then:

```
assert serialized_fields == expected_fields, \
    "Field specified in mapping should be included with default value"
```

After this second test, the branch coverage has improved to 100%. The command to see the code using `git diff` (from the `improve-the-test-coverage` branch) is:

```
git diff e37c8bc82 1a2ab33a7 -- ./tests/test_exporters.py
```

The final coverage report can be seen on Fig. 3.

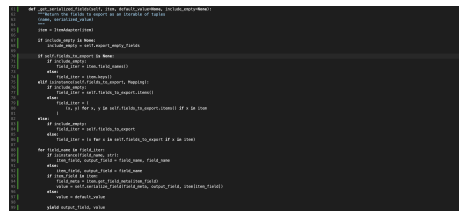


Figure 3: Coverage after Test no. 2 - `_get_serialized_fields`

5.1.6 Additional notes

- The function can be called directly in the test suite through the class it belongs to.
- It was easier to write new tests than to expand on existing tests.
- We didn't need to add additional interfaces.

5.2 `_link_allowed` function and Link testing

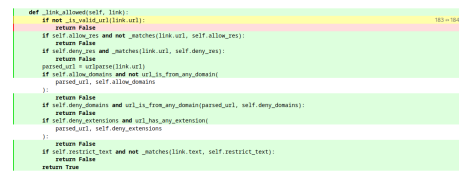
5.2.1 Initial coverage

For the `_link_allowed`, the initial coverage had one branch not covered, as found by using `coverage` on the code, so it has quite a high coverage already. Therefore, another function was also found, the `link` class which is used by the `_link_allowed` function. This function also had two branches which were not covered by tests. The initial coverage can be seen in Figure 4 and 5.



```
scrapy/link.py:74% 35 43 0 2
21
22 :param fragment: the part of the url after the hash symbol. From the sample, this is "foo".
23
24 :param nofollow: an indication of the presence or absence of a nofollow value in the "rel" attribute
25 of the anchor tag.
26 """
27
28 __slots__ = ["url", "text", "fragment", "nofollow"]
29
30 def __init__(
31     self, url: str, text: str = "", fragment: str = "", nofollow: bool = False
32 ):
33     if not isinstance(url, str):
34         got = url.__class__.__name__
35         raise TypeError(f"Link url must be str objects, got {got}")
36     self.url: str = url
37     self.text: str = text
38     self.fragment: str = fragment
39     self.nofollow: bool = nofollow
40
41 def __eq__(self, other: Any) -> bool:
42     if not isinstance(other, Link):
43         raise NotImplementedError
44     return (
45         self.url == other.url
46         and self.text == other.text
47         and self.fragment == other.fragment
48         and self.nofollow == other.nofollow
49     )
50
51 def __hash__(self) -> int:
52     return (
53         hash(self.url) ^ hash(self.text) ^ hash(self.fragment) ^ hash(self.nofollow)
54     )
55
56 def __repr__(self) -> str:
57     return (
58         f"Link(url={self.url!r}, text={self.text!r}, "
59         f"fragment={self.fragment!r}, nofollow={self.nofollow!r})"
60     )
```

Figure 4: Coverage Before for link class



```
def _link_allowed(link: Link):
    if not isinstance(link.url, str):
        return False
    if link.allow_res and not _matches(link.url, link.allow_res):
        return False
    if link.deny_res and _matches(link.url, link.deny_res):
        return False
    parsed_url = urlparse(link.url)
    if link.allow_domains and not url_is_from_any_domain(
        parsed_url, link.allow_domains
    ):
        return False
    if link.deny_domains and url_is_from_any_domain(parsed_url, link.deny_domains):
        return False
    if link.allow_extensions and not _has_any_extensions(
        parsed_url, link.allow_extensions
    ):
        return False
    if link.deny_extensions and _has_any_extensions(
        parsed_url, link.deny_extensions
    ):
        return False
    if link.restrict_text and not _matches(link.text, link.restrict_text):
        return False
    return True
```

Figure 5: Coverage before for `_link_allowed`

5.2.2 Identified requirements

For `_link_allowed`, the requirements that are found are

- **Req 1:** Link must be a valid url (must start with http or similar)
- **Req 2-3:** Link must respectively must not contain the allowed respectively not allowed characters
- **Req 4-5:** The domain is checked to be in the allowed/not allowed domains if any are given
- **Req 6:** Checks if url has extensions, only allow if extensions are allowed
- **Req 7:** Check if the text contains any restricted characters

For the link class, we instead have in the `__init__` and `__eq__` functions that

- **Req 1:** Url must be a string
- **Req 2:** The object we compare a link to must be a link
- **Req 3:** When comparing two links, they are the same if all attributes are the same.

5.2.3 Untested requirements

For `_link_allowed`, only requirement 1 is untested. For the `link` class, we have that requirement 1 and 2 is untested.

5.2.4 Tests on `_link_allowed`

The test on `_link_allowed` test that the extracted links is an empty list when given a non-valid link. The test `test_not_valid_link` is created in the file `scrapy/tests/test_linkextractors.py`. We give the link `test.test`, and check that

```
self.assertEqual(lx.extract_links(response), [])
```

The code can be found by running

```
git diff 6b2696bd9 8a7d80ea0 -- ./tests/test_linkextractors.py
```

Additionally, the coverage of `_link_allowed` after implementing this function is found in Figure 6. As we can see, the cases are fully covered after implementing the new tests.

5.2.5 Tests on Link class

Now, we look at the test added for the link class. The tests were added in `scrapy/tests/test_link.py` and cover one of the un-tested cases each. In the first one, `test_if_string`, we check that the function raises a `TypeError` when an integer is given instead of a string. Another test, `test_equals_not_link` checks that we get a `NotImplementedError` if we try to compare a link object with a non-link object, more specifically an integer.

The code can be found using The code can be found by running


```

181
182 def _link_allowed(self, link):
183     if not _is_valid_url(link.url):
184         return False
185     if self.allow_res and not _matches(link.url, self.allow_res):
186         return False
187     if self.deny_res and _matches(link.url, self.deny_res):
188         return False
189     parsed_url = urlparse(link.url)
190     if self.allow_domains and not url_is_from_any_domain(
191         parsed_url, self.allow_domains
192     ):
193         return False
194     if self.deny_domains and url_is_from_any_domain(parsed_url, self.deny_domains):
195         return False
196     if self.deny_extensions and url_has_any_extension(
197         parsed_url, self.deny_extensions
198     ):
199         return False
200     if self.restrict_text and not _matches(link.text, self.restrict_text):
201         return False
202     return True
203

```

Figure 6: Coverage after tests for `_link_allowed` function

```
git diff 6b2696bd9 44a3a7623 -- ./tests/test_link.py
```

Also, in Figure 7, the coverage after the tests is shown.

```

27
28 __slots__ = ["url", "text", "fragment", "nofollow"]
29
30 def __init__(
31     self, url: str, text: str = "", fragment: str = "", nofollow: bool = False
32 ):
33     # Requirement 1:
34     # url must be a string
35     # Not tested
36     if not isinstance(url, str):
37         got = url.__class__.__name__
38         raise TypeError(f"Link urls must be str objects, got {got}")
39
40     self.url: str = url
41     self.text: str = text
42     self.fragment: str = fragment
43     self.nofollow: bool = nofollow
44
45 def __eq__(self, other: Any) -> bool:
46     # Requirement 2:
47     # object we compare to must be a Link
48     # Not tested
49     if not isinstance(other, Link):
50         raise NotImplementedError
51
52     # Requirement 3:
53     # Attributes must be the same
54     # Tested
55     return (
56         self.url == other.url
57         and self.text == other.text
58         and self.fragment == other.fragment
59         and self.nofollow == other.nofollow
60     )

```

Figure 7: Coverage after tests for link class

5.2.6 Additional notes

The `_link_allowed` function is called through the `extract_link`-function, so it was tested through that function. The `Link` class can be called directly. No additional interfaces were needed, and new tests were written instead of expanding existing ones.

5.3 parse.Command.max_level

There was no coverage for the property `max_level` in the `parse.Command` class. So tests had to be implemented to test the 5 different possible branches.

5.3.1 Identified Requirements

- Correct function if `self.items` and `self.requests` are non-empty dictionaries.
- Correct handling if `self.items` is empty.
- Correct handling if `self.requests` is empty.
- Correct handling if both `self.items` and `self.requests` are empty.

5.3.2 Untested Requirements

Every requirement was untested in the original test suite. Therefore, tests were written to cover all requirements.

5.3.3 Test

A test was added that tested correct functionality if self.items and self.requests were two dictionaries, if requests or items were empty and if both were empty.

This yielded full branch coverage for the max_level function. <https://www.overleaf.com/project/65cf53617b7e9fe>

5.3.4 Additional notes

- The function can be called directly in the test suite through the class it belongs to.
- It was easier to write new tests than to expand on existing tests.
- We didn't need to add additional interfaces.

5.4 _get_inputs

5.4.1 Initial coverage

For the _get_input, there are 12 coverage and test cases cover 11/12, so initial coverage is 92%. See image below for details.

```
def _get_inputs(
    form: FormElement,
    formdata: FormDataType,
    dont_click: bool,
    clickdata: Optional[dict],
) -> List[FormDataKVType]:
    """Return a List of key-value pairs for the inputs found in the given form."""
    try:
        formdata_keys = dict(formdata or {}).keys()
    except (ValueError, TypeError):
        raise ValueError("formdata should be a dict or iterable of tuples")

    if not formdata:
        formdata = {}
    inputs = form.xpath(
        "descendant::testarea"
        "descendant::select"
        "descendant::input[not(@type or @type='')]"
        "not(ancestor::input[reset])", "1")
    = and (...)@checked or"
    ' not(ancestor::input[radio])', "1")]]],
    namespaces={"re": "http://exslt.org/regular-expressions"},
)
values: List[FormDataKVType] = [
    (k, "" if v is None else v)
    for k, v in ((value(e) for e in inputs)
                 if k and k not in formdata_keys
                 )
]
if not dont_click:
    clickable = _get_clickable(clickdata, form)
    if clickable and clickable[0] not in formdata and not clickable[0] is None:
        values.append(clickable)

if isinstance(formdata, dict):
    formdata = formdata.items() # type: ignore[assignment]
values.extend((k, v) for k, v in formdata if v is not None)
return values
```

Figure 8: Coverage Before for _get_input function

5.4.2 Identified Requirements

- Req. 1: formdata must be dict type
- Req. 2: If formdata is empty, assign an empty list

- **Req. 3:** Retrieve the results of a search for elements in an XML or HTML document using an XPath query, retrieve the results in order, and assign elements with keys that are not in `formdata_key` to a new array `values`
- **Req. 4 :** Determine if it is clickable or not, and if it is found to be clickable and a non-empty result is returned that is not in the `formdata`, add it to the `values`.
- **Req. 5 :** If `formdata` exists as dict type, only items of `formdata` are assigned to `formdata`.

5.4.3 Untested Requirements

Req. 1 are not tested in the current test cases.

5.4.4 Test

Without refactoring of `_get_input` and `form.py`, it is impossible to meet the requirements of **Req. 1** (it is effectively dead code). Therefore, we will not add the test in this chapter, but will discuss it in detail in the factoring chapter.

5.4.5 Additional notes

- The function can be called indirectly in the test suite through the class it belongs to.
- It was easier to write new tests than to expand on existing tests.
- We didn't need to add additional interfaces.

6 Refactoring

Here we lay out our refactoring plans for 5 complex functions in the code.

6.1 `_get_serialized_fields` function

We describe here the refactoring plan for the `_get_serialized_fields` function, implemented as a method of the `BaseItemExporter` class in the `scrapy/scrapy/exporters.py` module.

6.1.1 Step 1 : identify functional blocks

In the `_get_serialized_fields` function, the two main functional blocks are:

- determining `field_iter` based on conditions
- iterating through fields and processing each one

6.1.2 Step 2 : simplify the function using helper methods

Having identified the functional blocks, we can break down the function into more manageable chunks that each deal with a different aspect of the functionality. Furthermore, it improves separation of concerns.

- Helper method no. 1: `_determine_field_iter` - This first method makes a decision based on `include_empty`, `self.fields_to_export`, and in particular whether `self.fields_to_export` is a mapping.
- Helper method no. 2: `_process_field` - The purpose of this second method is to process a single field, so the `_get_serialized_fields` can just call it multiple times to process fields.

6.1.3 Step 3: update the `_get_serialized_fields` function

Now we can make use of the helper methods in the main method to achieve the same functionality but with a less complex function. It would now look like this:

```
def _get_serialized_fields(self, item, default_value=None, include_empty=None):
    item = ItemAdapter(item)
    include_empty = self.export_empty_fields if include_empty is None else include_empty
    field_iter = self.determine_field_iter(item, include_empty)
    for field_name in field_iter:
        yield self.process_field(item, field_name, default_value)
```

It is much shorter and the complexity is now reduced to 3 (count by hand).

6.1.4 Step 4 : update unit tests

We would now need to update the test suite to cover the newly created methods individually, in order to ensure that each unit works as expected. Then we would conduct thorough testing to ensure that the refactored code behaves identically to the original code.

6.1.5 Step 5 : documentation and peer review

Finally, we would need to document the refactoring (like I am doing right now), explaining why it was done and how, but also include relevant docstrings and comments in the newly created functions. Then, before merging the new version of the code, it should go through peer review as usual.

6.1.6 Additional notes

Here the refactoring of this function is very beneficial because it reduces greatly its cyclomatic complexity, without making the code much more complicated to understand. Indeed, separation of concerns makes sense and the two helper functions are not overly complicated. There are no drawbacks in this case.

6.2 `_link_allowed` function

Here, the refactoring of the `_link_allowed` function in the `LxmlLinkExtractor` class is described in the `scrapy/linkextractors/lxmlhtml.py` file.

6.2.1 Step 1: Identify parts of the code

Looking at the code, we can see that it is structured as a number of if-clauses with conditions a link should follow. If it does not fulfill one of the criteria, the function returns false. One way to split this into blocks would be to divide these if-clauses into categories depending on the condition they test. For example, one could create the categories

- deny or allow url depending on its content
- deny or allow domain for the link

6.2.2 Step 2: Create helper methods

Now, we can use this division of the code to create two helper methods, one for each of these cases

- `check_url` - does the checks on the contents of the url compared to the allowed contents
- `check_domain` - does the checks on if the domain is an allowed or denied domain.

6.2.3 Step 3: update `_link_allowed` function

We then get the three of the if-clauses from before (checking if the url is valid, if there is any extensions and if the text is restricted), as well as two if statements checking the result of the helper functions.

Then, the complexity is (counted by hand)

$$M = \pi - s + 2 = 7 - 6 + 2 = 3$$

which is lower.

6.2.4 Step 4: Update unit tests

Tests of the new helper functions should be added, to make sure they give a correct result. Also, the whole program should be tested, to see that the changes haven't introduced any new bugs.

6.2.5 Step 5: Documentation

Lastly, the new methods need to be documented.

6.2.6 Additional notes

In this case, the refactoring does decrease the complexity of the function. However, it might also make it less clear what is happening, since the code for the helper methods will be separate from the function, and reading the code then requires more work. The function is, despite its somewhat high complexity, pretty readable to start with, and therefore this refactoring might not be worth it.

6.3 `_processor_spider_output`

Refactoring plan for the `_processor_spider_output` function in `scrapy/scrapy/core/spidermw.py`.

6.3.1 Step 1: identify functional blocks

There are two code blocks than can be broken out.

- Determine what method to use depending on if it is run sync or async.
- Handle the upgrade/downgrade if needed.

6.3.2 Step 2 : simplify the function using helper methods

These code blocks can now be broken out into helper functions that only deal with these particular smaller problems and can be called on from the original function.

- `_process_helper()` - This function could determine what method to use (async or sync) depending on what is stored in the `method_pair` variable and return the correct method.
- `_method_execution_helper()` - This function could take in a method and if it needs upgrading or downgrading it returns the correct version of the method.

6.3.3 Adding unit tests

Some additional unit tests should be added to cover the helper functions to make sure overall coverage is not reduced and to make sure that they function correctly.

6.3.4 Additional comments

Reducing complexity by refactoring code in to smaller functions can make the code less readable, but in this case the pros would most likely outweigh this con, as both parts of the code that was broken out is expected to be removed when they drop compatibility for downgrading and async. Having this code in smaller functions might make this process of removing it easier.

6.4 `_get_input`

Refactoring plan for the `_get_input` function in `scrapy/http/request/form.py`.

6.4.1 Step 1: Identify functional blocks

As mentioned in the COVERAGE chapter, the Req. 1 section is creating DEAD CODE. By outsourcing this type checking to an external function, we can reduce the complexity of the `_get_input` function without causing errors.

6.4.2 Step 2 : Create a helper method

Specifically, in the `__init__` function of the `FormRequest` class, if the `formdata` is not a dict type, an error occurs when executing the `_urlencode` function, which also exists in `form.py`. Therefore, we can introduce a type-checking try-except function in the `_urlencode` function.

6.4.3 Step 3: update `_get_input` function and adding test cases

It is necessary to check if `TypeError` is output from `_urlencode`. By doing this and removing the **Req. 1** process from `_get_input`, we can effectively increase the branch coverage to 100%. In addition, the number of branches and the circulation complexity can be reduced by 2 each. In testing, two additional test cases were added. Details are as follows.

- `test_formdata_is_list()` - Assign any list-type array method.
- `test_formdata_is_tuple()` - Assign any tuple type array

7 Self-assessment: Way of working

We still feel like our way of working fulfills the “In place” level of the checklist. The tools that we are using, for example GitHub issues and pull requests, are still working well. This time, since more of the work is focused on different tasks than just writing code, we have been using GitHub to a somewhat smaller extent. We still have issues for the tasks that need to be done, but there has been less work on for example peer reviewing code. Overall, since the start of the course, we have improved our communication over what needs to be done, as well as become more comfortable with using GitHub for organizing the work.

To get to the “Working well”-level, we still need to become more comfortable with the way of working that we are using. We would also need to do even more evaluating and adjusting of how we are using the tools we have to best help us work together. For example, the next lab has a similar structure to this one, so before starting that one, we need to discuss what worked well and what did not work well whilst doing this lab.

8 Overall experience

In this project, we learned about software testing, specifically focusing on cyclomatic complexity, test coverage, and code refactoring. We discovered the importance of test coverage tools and their role in identifying untested code parts, leading to more reliable software. Refactoring was key to reducing code complexity, making it cleaner and easier to understand. We faced challenges in integrating tools and highlighted the importance of documentation and teamwork. All in all, this experience enhanced our understanding of open-source software.