Info3 - Lab 07 - Testing

Group: Aaron Rau, Jonas Trenkler
Repository: https://github.com/JonasTrenkler/info3-lab07-testing

1. Getting Started

Closed-Boxed Testing and Analysis

Closed-box tests only have the interface to work with. A method that returns the absolute of an integer should have a simple signature like `absolute(i)`, with only a single parameter, expected to be an integer. As Python is a dynamically typed language there is no guarantee that the caller respects this limitation. While using type-hints helps to reduce errors while writing the code, the method is still callable during runtime.

Therefore this is one equivalence class: *Invalid input* or `NaN` (though not `math.nan` or `float("nan")`). The implementation has to decide how to handle this if there is no specification. One reasonable way to handle invalid inputs would be to throw a `TypeError`. Another less obvious way would be to use error codes, for example negative integers. The invalid input could also just pass through unmodified.

Apart from that there are numbers, not differentiating between integers and floating point numbers in the following equivalence classes:

- below 0
- above 0
- 0

Treating 0 differently might not be necessary, but it is certainly a special case worth to look at. The floating point numbers in Python adhere to IEEE 754, which does define both positive and negative zero, see Signed Zero (Wikipedia). The negative zero behaves like a negative number, for example in multiplication, but it is equal to 0 in a boolean statement. Therefore the following is possible in python:

```
In [7]: 0.0 * 2
Out[7]: 0.0

In [8]: -0.0 * 2
Out[8]: -0.0

In [9]: 0.0 * -2
Out[9]: -0.0

```

```
In [10]: 0.0 == -0.0
Out[10]: True
```

From the above we could argue that there are three equivalence classes:

1. numbers from 0.0 to positive infinity
2. numbers from -0.0 to negative infinity
3. invalid inputs, NaN

Open-Box Tests and Source Code Analysis

The method only has for lines of code and a singe if-else, resulting in two branches to be tested:

```
def absolute_value_of(x):
    if x < -1:
        return -x
    else:
        return x
```

By just looking at the code, a kind of static code analysis, it is obvious that the condition `x < -1` will not return the absolute value of −1 correctly.

The module Coverage can generate coverage reports. It is also accessible using the pytest-cov plugin. But since `pytest-cov` can only test modules, it will provide coverage data for the `tax_time` module as well:

```
╰─

 pytest --cov-report term-missing --cov=python tests/test_absolute.py
    --cov-branch

======================== test session starts =========================
   platform linux -- Python 3.11.3, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/jonas/Studium/B15_Info3/labs/info3-lab07-testing/
   a_open_and_closed_box_tests
plugins: cov-4.1.0
collected 0 items

---------- coverage: platform linux, python 3.11.3-final-0 -----------
Name                     Stmts   Miss Branch BrPart  Cover   Missing
--------------------------------------------------------------
python/absolute.py           4      3      2      0    17%   3-6
python/tax_time.py          46     46     14      0     0%   2-65
--------------------------------------------------------------
TOTAL                       50     49     16      0     2%

======================= no tests ran in 0.02s ========================
```

Using the `coverage` CLI directly, two calls are necessary, one to generate the coverage data and one to display the report. But this way, the `include` parameter is available, which let's us limit the report to the file `absolute.py` (see SO issue)

```
╰
coverage run --branch -m pytest tests/test_absolute.py -v &&
      coverage report --include=python/absolute.py
```

To cover all branches, two tests are sufficient.

```python
from python.absolute import absolute_value_of

def test_1():
    assert absolute_value_of(1) == 1

def test_minus_2():
    assert absolute_value_of(-2) == 2
```

```
╰
coverage run --branch -m pytest tests/test_absolute.py -v &&
      coverage report --include=python/absolute.py

======================== test session starts =========================
   platform linux -- Python 3.11.3, pytest-7.4.0, pluggy-1.2.0 -- /home
   /jonas/.local/share/virtualenvs/info3-lab07-testing-z5eA380H/bin/
   python
cachedir: .pytest_cache
rootdir: /home/jonas/Studium/B15_Info3/labs/info3-lab07-testing/
   a_open_and_closed_box_tests
plugins: cov-4.1.0
collected 2 items

tests/test_absolute.py::test_1 PASSED                           [ 50%]
tests/test_absolute.py::test_minus_2 PASSED                     [100%]

========================= 2 passed in 0.02s ==========================
Name                   Stmts   Miss Branch BrPart  Cover
--------------------------------------------------------
python/absolute.py         4      0      2      0   100%
--------------------------------------------------------
TOTAL                      4      0      2      0   100%
```

While these two tests both pass and cover all branches, they to not reveal the error in the method. Testing the edge cases, so one value above and below the conditions, yields the following tests for the number equivalence classes positive and negative integers.

```
╰
coverage run --branch -m pytest tests/test_absolute.py -vv &&
      coverage report --include=python/absolute.py
```

```

======================== test session starts =========================
platform linux -- Python 3.11.3, pytest-7.4.0, pluggy-1.2.0 -- /home/jonas/.local/share/virtualenvs/info3-lab07-testing-z5eA380H/bin/python
cachedir: .pytest_cache
rootdir: /home/jonas/Studium/B15_Info3/labs/info3-lab07-testing/a_open_and_closed_box_tests
plugins: cov-4.1.0
collected 5 items

tests/test_absolute.py::test_1 PASSED                                [ 20%]
tests/test_absolute.py::test_0 PASSED                                [ 40%]
tests/test_absolute.py::test_negative_0 PASSED                       [ 60%]
tests/test_absolute.py::test_negative_1 FAILED                       [ 80%]
tests/test_absolute.py::test_negative_2 PASSED                       [100%]

============================== FAILURES ==============================
__________________________ test_negative_1 ___________________________
    def test_negative_1():
>       assert absolute_value_of(-1) == 1
E       assert -1 == 1
E        +  where -1 = absolute_value_of(-1)

tests/test_absolute.py:13: AssertionError
====================== short test summary info =======================

FAILED tests/test_absolute.py::test_negative_1 - assert -1 == 1

==================== 1 failed, 4 passed in 0.12s =====================
```

As predicted, the test for -1 failed. Blindly writing open-box tests for all branches, or line coverage does not necessarily reveal problematic code. It is necessary to think about the expected input and output of the method, the equivalence classes and edge cases as well. Tests for the equivalence class of invalid input are not really feasible without an agreement how this should be handled.

2. Test Driven Development

With the knowledge of how a linked list works and the already existing tests the implementation was quite easy. It's hard to say though if this approach is quicker than the traditional one, since the tests were already there in this case. If the tests had to be written as well, it might have been more overhead.

The only standard library dependency of this implementation is the Self class from the `typing` module, but that wouldn't be necessary either. The copy method could have been solved by using a function from the standard library.

```python
from copy import copy

def copy(self) -> Self:
    return copy(self)
```

But since that is rather boring the final implementation looks like this:

```python
 def copy(self) -> Self:
    copy = self.shallow_copy()
    curr = self.next

    while curr is not None:
        copy.append(curr.data)
        curr = curr.next

    return copy
```

`shallow_copy` constructs a new `Node` that only contains the `data` of the current object, not the next node.

The main idea behind most of the methods is a while loop that checks if either the current or the next node is not `None`. The reason for not all of the loops using the same condition is that some methods need to handle the first node differently. The general pattern looks like this:

```python
curr: Node = self

while curr is not None:
    # do some stuff
    curr = curr.next
```

Even though all of the tests are green, there might still be some bugs. The `delete` method, for example, immediately returns self.next if the nodes data matches the input data. If deleting all nodes if that name is the desired behaviour it would work correctly. Likewise, if the expected behaviour is that only the first match is deleted the implementation would also cause issues.

One more thing to note is that the `from_str` function strips the spaces from each element that is to be added. This means that `'a,b'` and `'a, b'` are equivalent. This would cause issues if trailing spaces are expected to hold meaning.

Generally speaking it is not clear whether or not test driven development improves productivity and maintainability and reduces the number of bugs in the final implementation. Working with the already existing tests felt pretty efficient though.