Trevor White (20348221)
Sophia Pagazani (20333850)
Aniss Hamouda (20348807)
CISC 327
11/11/2024

Project Assignment 4
Code Coverage Analysis

1. Initial Code Coverage

Using the suggested `coverage.py`, which calculates code coverage for a set of Python test scripts, the below coverage report was generated. The corresponding `index.html` can be found under `Assignment-4/initial_coverage`. On the initial pass on the test suites, our project had a coverage of 95%. There were only 6 files that did not have full coverage. Two of these did not meet the target criteria of 90% or above—`app.py` and `database.py`—while the other four did. In the next section, we determine why each missing line was not executed and how to appropriately increase coverage.

Coverage report: 95%

Files | Functions | Classes

coverage.py v7.6.4, created at 2024-11-06 15:18 -0500

File ▲	statements	missing	excluded	coverage
app.py	59	14	0	76%
database.py	7	1	0	86%
tests/__init__.py	0	0	0	100%
tests/test_client_by_name.py	13	0	0	100%
tests/test_client_filter.py	61	5	0	92%
tests/test_client_lookup.py	53	0	0	100%
tests/test_drug_filter.py	64	5	0	92%
tests/test_drug_lookup.py	43	0	0	100%
tests/test_get_set_prescriptions.py	63	0	0	100%
tests/test_prescription_creation.py	30	0	0	100%
tests/test_save_new_prescription.py	31	0	0	100%
tests/test_valid_drug.py	17	0	0	100%
tests/test_valid_input.py	30	1	0	97%
tests/test_view_prescriptions.py	12	0	0	100%
utility.py	94	2	0	98%
Total	**577**	**28**	**0**	**95%**

coverage.py v7.6.4, created at 2024-11-06 15:18 -0500

Figure 1: Initial code coverage report

2. Missed Statements and Actions Taken

2.1 app.py

This file is one of the two that did not meet the 90% coverage condition, with only 78% total statement coverage. The screenshots in Figure 2 clearly indicate which lines failed to be executed. We take a detailed look at each of these lines.

Coverage for **app.py**: 76%

59 statements | 45 run | 14 missing | 0 excluded

« prev ^ index » next coverage.py v7.6.4, created at 2024-11-06 15:18 -0500

```
1   """
2   This program implements a medicine management website. Users can
3   search a database of 20 drugs, and another of 20 clients. They
4   can also view a client's prescriptions, modify their status, and
5   create new prescriptiosn.
6
7   This file provides the main control flow of the app. It bridges between
8   the backend link to the database (built with MongoDB) and the front-end UI
9   (a combination of html/css/js).
10
11  To run the app, if the project is setup correctly, simply execute
12  'python app.py' in the app/ directory. For more detailed information
13  about setup and running the app, please see the README file under the
14  Assignment-3/ directory.
15  """
16
17  from flask import Flask, render_template, request, jsonify
18  import utility
19
20  app = Flask(__name__)
21
22  # Launch Page-----------------------------------------------------
23  @app.route('/', methods=['GET', 'POST'])
24  def home():
25      return render_template('home.html', user_input=None)
26
27
28  # Clients Search Page---------------------------------------------
29  @app.route('/clients', methods=['GET', 'POST'])
30  def clients():
31      # Backend link to get the data.
32      data = utility.get_all_clients()
33
34      # Filter data set according to user's search parameters.
35      if request.method == 'POST':
36          try:
37              par = request.form['search_par']
38          except:
39              par = None
40          data = utility.filter_clients(data, par)
41
42      # Renders the page.
43      return render_template('clients_search.html', data=data)
44
```

```
45  # Clients Information Page----------------------------------------
46  @app.route('/clients/<phone_number>', methods=['GET', 'POST'])
47  def clients_info(phone_number):
48      # Backend link to get the data about the client.
49      client = utility.get_client_by_phone(phone_number)
50      # Renders the page.
51      return render_template('clients_info.html', client=client)
52
53  # Clients Prescription Creation Page-----------------------------------
54  @app.route('/clients/<phone_number>/create', methods=['GET', 'POST'])
55  def clients_create(phone_number):
56      if request.method == 'POST':
57          try:
58              # Validate input
59              user_input = [int(request.form['DIN']),
60                            request.form['preBy'],
61                            request.form['refill']]
62              entry = utility.valid_drug(user_input[0])
63
64              # Save to database and redirect.
65              if(entry != None):
66                  utility.save_new_prescription(phone_number, user_input, entry)
67                  return clients_info(phone_number)
68
69              # Throw alert if drug does not exist in the database.
70              else:
71                  return render_template('clients_create.html', phone_number=phone_number, alert=True)
72          # Throw alert if DIN is not an integer.
73          except:
74              return render_template('clients_create.html', phone_number=phone_number, alert=True)
75      # Renders the page.
76      return render_template('clients_create.html', phone_number=phone_number, alert=False)
77
78
79  # Inventory Search Page-------------------------------------------
80  @app.route('/inventory', methods=['GET', 'POST'])
81  def inventory_search():
82      # Backend link to get the data.
83      data = utility.get_all_inv()
84
85      # Filter data set according to user's search parameters.
86      if request.method == 'POST':
87          try:
88              par = request.form['search_par']
89          except:
90              par = None
91          data = utility.filter_inv(data, par)
92
93      # Renders the page.
94      return render_template('inventory_search.html', data=data)
95
96  # Inventory Information Page -------------------------------------------
97  @app.route('/inventory/<din>', methods=['GET', 'POST'])
98  def inventory_info(din):
99      return render_template('inventory_info.html', din=din)
100
101
102 # Supply Order View Page-----------------------------------------
103 @app.route('/supply', methods=['GET', 'POST'])
104 def supply():
105     return render_template('supply.html', user_input=None)
106
107 ############################################################
108 @app.route('/update_list', methods=['POST'])
109 def update_lists():
110     data = request.get_json() # retrieve the data sent from JavaScript
111     # process the data using Python code
112     phone = data['id']
113     actives = data['active']
114     olds = data['old']
115
116     utility.update_prescriptions(phone, actives, olds)
117     return jsonify(result=phone)
118
119 if __name__ == '__main__':
120     app.run(debug=True)
```

Figure 2: Missed statements in app.py

First, notice line 120. This line launches the main program in debug mode when the `python app.py` command is run in the terminal. Due to the way the testing tool generates the "dummy" webpage instances, this file is never called as the main program, so it is expected that this line is not covered.

Next, we also notice a few lines with `return render_template()` that do not get triggered by the test suite: lines 25, 99, and 105, corresponding to the home page, the inventory information page, and the supply order page, respectively. The latter two did not have tests created for those pages as they were not part of the intended implemented functionality, while the former was an oversight. A new test called `test_misc_pages.py` was added for these three miscellaneous pages.

Furthermore, the `expect` blocks for two `try-except` statements found at lines 38, 39, 98, and 99 were skipped as well. These blocks intend to catch when user input is empty on the search pages. As it turns out, they are never entered because `Flask` receives empty strings instead of the `None` value! Thus, these statements can simply be removed.

Finally, the function `update_list()` in `app.py` (lines 110-117 in Figure 2) was not tested thoroughly due to its deep connections with the update prescription feature (which consists of a lot of work in Javascript as opposed to Python, the language we are testing in). It was seen as too complicated; however, a closer look revealed that it could be tested in isolation from the frontend, so a test was added, as seen in section 3 in this report.

2.2 `database.py`

This is our second file with a coverage percentage less than 90%, only covering 86% of the lines. We can see in Figure 3 that only one of the statements is missing from our tests, and since the module has a low amount of statements, it has affected the coverage in a disproportionate way.

Figure 3: Missed statements in `database.py`

During further inspection, we noticed that not only does this line not get tested, it does not run at execution, nor is it used by any other function. It was added when following a tutorial on how to set up the database but was not relevant to any other sections of our codebase. On that reasoning, we opted to remove lines 17-20, as writing a test for code with no purpose would be unproductive.

2.3 `test_client_filter.py`

For `test_client_filter.py`, there were 5 lines that were not covered, all for the same reason. These are shown in Figure 4. They exist in order to catch when the tests fail, but since the tests currently do not fail, the lines are not covered. The coverage percentage for this file still meets the 90% threshold, so we did not change this file.

```
67  # Failure cases for filter_clients() ------------------------------
68  def test_filter_clients_fail_1():
69      """Failure case: filter_clients() throws TypeError if clients is not iterable."""
70      try:
71          utility.filter_clients(0, "a")
72          assert False, "Did not throw TypeError for clients not iterable"
73      except TypeError:
74          assert True
75
76  def test_filter_clients_fail_2():
77      """Failure case: filter_clients() throws TypeError if clients[entry] is not indexable"""
78      try:
79          utility.filter_clients([1], "a")
80          assert False, "Did not throw TypeError for clients[entry] not indexable"
81      except TypeError:
82          assert True
83
84  def test_filter_clients_fail_3():
85      """Failure case: filter_clients() throws IndexError if clients[entry][0] does not exist"""
86      try:
87          utility.filter_clients([[]], "a")
88          assert False, "Did not throw IndexError for clients[entry][0] does not exist"
89      except IndexError:
90          assert True
91
92  def test_filter_clients_fail_4():
93      """Failure case: filter_clients() throws AttributeError if clients[entry][0] is not a string"""
94      try:
95          utility.filter_clients([[0]], "a")
96          assert False, "Did not throw AttributeError for clients[entry][0] not a string"
97      except AttributeError:
98          assert True
99
100 def test_filter_clients_fail_5():
101     """Failure case: filter_clients() throws IndexError if clients[entry][2] does not exist"""
102     try:
103         utility.filter_clients([["a","b"]], 0)
104         assert False, "Did not throw IndexError for clients[entry][2] does not exist"
105     except IndexError:
106         assert True
107
```

Figure 4: Missed statements in `test_client_filter.py`

2.4 test_drug_filter.py

For test_drug_filter.py, the situation is the same as in section 2.3. The 5 lines that are not covered in Figure 5 exist in order to catch when the tests fail, but as it is, they are passing, so the lines are overlooked. As before, the coverage percentage for this file still meets the 90% threshold, so we did not change this file.

```
72  # Failure cases for filter_inv() ------------------------------
73  def test_filter_inv_fail_1():
74      """Failure case: filter_inv() throws TypeError if inv is not iterable."""
75      try:
76          utility.filter_inv(0, "a")
77          assert False, "Did not throw TypeError for inv not iterable"
78      except TypeError:
79          assert True
80
81  def test_filter_inv_fail_2():
82      """Failure case: filter_inv() throws TypeError if inv[entry] is not indexable"""
83      try:
84          utility.filter_inv([0], "a")
85          assert False, "Did not throw TypeError for inv[entry] not indexable"
86      except TypeError:
87          assert True
88
89  def test_filter_inv_fail_3():
90      """Failure case: filter_inv() throws IndexError if inv[entry][0] does not exist"""
91      try:
92          utility.filter_inv([[]], "a")
93          assert False, "Did not throw IndexError for inv[entry][0] does not exist"
94      except IndexError:
95          assert True
96
97  def test_filter_inv_fail_4():
98      """Failure case: filter_inv() throws AttributeError if inv[entry][0] is not a string"""
99      try:
100         utility.filter_inv([[0]], "a")
101         assert False, "Did not throw AttributeError for inv[entry][0] not a string"
102     except AttributeError:
103         assert True
104
105 def test_filter_inv_fail_5():
106     """Failure case: filter_inv() throws IndexError if inv[entry][1] does not exist"""
107     try:
108         utility.filter_inv([[0]], 0)
109         assert False, "Did not throw IndexError for inv[entry][1] does not exist"
110     except IndexError:
111         assert True
112
```

Figure 5: Missed statements in test_drug_filter.py

2.5 test_valid_input.py

For test_valid_input.py, the only issue was the first test case (see Figure 6). We realized that this case was poorly designed and should always fail. Up until now, however, it had gone unnoticed because it had the same name as another case, causing it to be shadowed. Thus, we removed the misleading test from the file to reduce confusion and for the sake of completion, even though it already met the 90% coverage threshold.

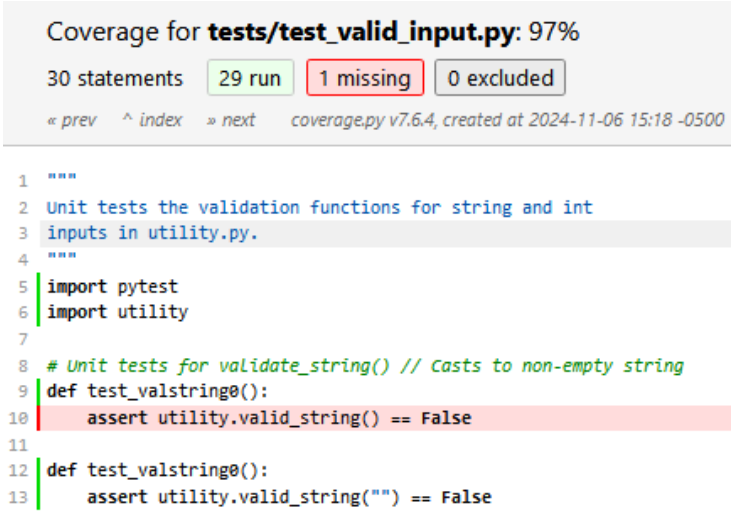

Figure 6: Missed statements in `test_valid_input.py`

2.6 `utility.py`

For `utility.py`, there were only 2 such lines, both of which were the return statements following a check that the input to `valid_string()` and `valid_int()` was not `None`. This follows from our observations in section 2.1, wherein we noticed that `None` was in fact not a possible return value of the form, meaning it is also not a possible input for these functions. Therefore, it does not make sense to include `None` in our input partitions for the respective test cases. Although not strictly necessary since coverage was greater than 90%, removing this pointless check was logical and was an easy way to further boost coverage.

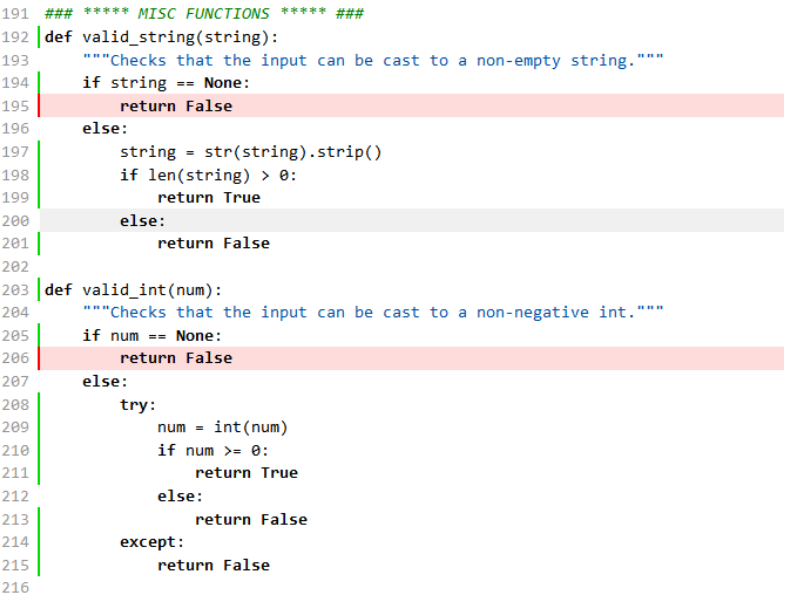

Figure 7: Missed statements in `utility.py`

3. New Test Scripts

As explained in the prior section, many of the coverage issues were indicative of unused code, which we removed. Only two new test scripts were required to bring coverage for all files above the desired 90% threshold. The two scripts in question are named `test_misc_pages.py` and `test_update_list.py`. The following figures show screenshots of the code written for these tests and their successful execution.

```
(venv327) trevorwh113@LAPTOP-R8J5CC1G:~/CISC327-G8/app/tests$ pytest test_misc_pages.py test_update_list.py
============================================= test session starts ==============================================
platform linux -- Python 3.10.12, pytest-8.3.3, pluggy-1.5.0
rootdir: /home/trevorwh113/CISC327-G8/app/tests
collected 4 items

test_misc_pages.py ...                                                                                 [ 75%]
test_update_list.py .                                                                                  [100%]

============================================== 4 passed in 1.74s ===============================================
(venv327) trevorwh113@LAPTOP-R8J5CC1G:~/CISC327-G8/app/tests$
```

Figure 8: Proof of success for the two new tests

```
"""
This file contains functional tests for the view of the home page
and two pages which are not included in our selected implemented
functionality: the supply order page and the drug information view
page. This test was added to ensure code coverage > 90%.
"""
from app import app
import pytest


def test_load_home_page():
    """
    Success case: Tests that the home page is loaded correctly (GET).
    """
    # Load the home page (GET)
    response = app.test_client().get('/')
    # Make sure correct page comes up.
    assert response.status_code == 200
    assert b"Medicine Management System" in response.data, "Did not load home page correctly"
    assert b"Manage Client Prescriptions" in response.data, "Did not load home page correctly"
    assert b"Search Inventory" in response.data, "Did not load home page correctly"
    assert b"Manage Supply Orders" in response.data, "Did not load home page correctly"

def test_load_inv_info_page():
    """
    Success case: Tests that the inventory information page is loaded correctly (GET).
    """
    # Load the inventory information page (GET)
    response = app.test_client().get('/inventory/904954')
    # Make sure correct page comes up.
    assert response.status_code == 200
    assert b"Error" in response.data, "Did not load inventory information page correctly"
    assert b"904954" in response.data, "Did not load inventory information page correctly"

def test_load_supply_page():
    """
    Success case: Tests that the supply page is loaded correctly (GET).
    """
    # Load the supply page (GET)
    response = app.test_client().get('/supply')
    # Make sure correct page comes up.
    assert response.status_code == 200
    assert b"Error" in response.data, "Did not load supply page correctly"
```

Figure 9: Implementation of `test_misc_pages.py`

```python
"""
This file contains a unit test for the update_list feature within the client info page
that switches prescription lists.
"""
from app import app
import pytest, utility

def test_update_list():

    c = utility.get_client_by_phone("(123)-456-7890")
    test_data = {
            'id': c[0],
            'active': utility.get_active_prescripts(c),
            'old': utility.get_old_prescripts(c)
    }

    # Load the update page (POST)
    response = app.test_client().post('/update_list', json=test_data)

    # Assert that the response is 200 OK
    assert response.status_code == 200

    # Assert the response JSON contains the expected result
    assert response.json == {'result': c[0]}
```

Figure 10: Implementation of `test_update_list.py`

4. Updated Code Coverage

Finally, we executed the coverage tool once more. The results are shown in Figure 11 and can be accessed under `Assignment-4/updated_coverage`. The total code coverage is now up to 98%, and no individual file is covered less than 90%. Notably, app.py rose from 76% to 98%.

Coverage report: 98%

Files | Functions | Classes

coverage.py v7.6.4, created at 2024-11-10 19:01 -0500

File ▲	statements	missing	excluded	coverage
app.py	53	1	0	98%
database.py	5	0	0	100%
tests/__init__.py	0	0	0	100%
tests/test_client_by_name.py	13	0	0	100%
tests/test_client_filter.py	61	5	0	92%
tests/test_client_lookup.py	53	0	0	100%
tests/test_drug_filter.py	64	5	0	92%
tests/test_drug_lookup.py	43	0	0	100%
tests/test_get_set_prescriptions.py	63	0	0	100%
tests/test_misc_pages.py	18	0	0	100%
tests/test_prescription_creation.py	30	0	0	100%
tests/test_save_new_prescription.py	31	0	0	100%
tests/test_update_list.py	8	0	0	100%
tests/test_valid_drug.py	17	0	0	100%
tests/test_valid_input.py	28	0	0	100%
tests/test_view_prescriptions.py	12	0	0	100%
utility.py	90	0	0	100%
Total	**589**	**11**	**0**	**98%**

Figure 11: Updated code coverage report