CS 472 Senior Design

Thien Van Ky Nguyen – Dynamic Analysis

Janurary 29th 2024

Fork repository: https://github.com/thienguen/barbell/

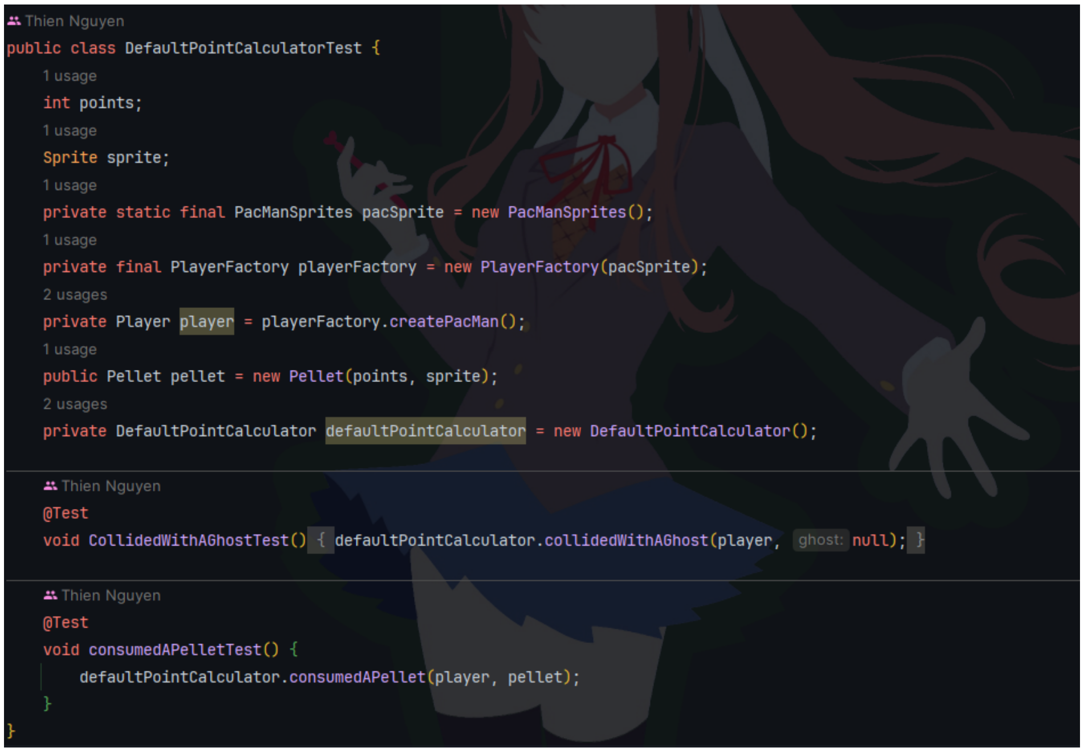
Task 1 - Is the coverage good enough?

Out of 55 classes, only 2 classes are covered. Out of 312 methods, only 5 methods are covered. Out of 1137 lines, only 14 lines are covered. With the statistic the answer is no, the coverage is not good. The coverage is only 3%. with the given tests that we have.

Task 2.1 - Increasing Coverage on JPacman

Method Paths

```
src/main/java/nl/tudelft/jpacman/points/PointCalculator.collidedWithAGhost
src/main/java/nl/tudelft/jpacman/points/PointCalculator.consumedAPellet
```

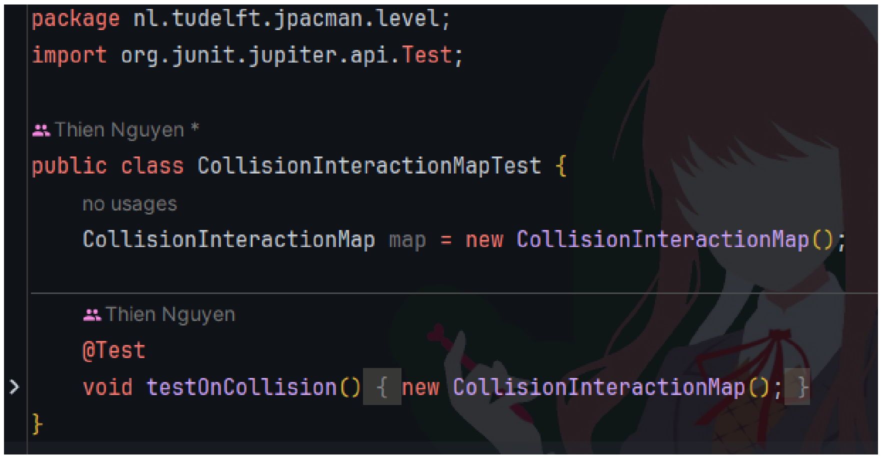


```
Thien Nguyen
public class DefaultPointCalculatorTest {
    1 usage
    int points;
    1 usage
    Sprite sprite;
    1 usage
    private static final PacManSprites pacSprite = new PacManSprites();
    1 usage
    private final PlayerFactory playerFactory = new PlayerFactory(pacSprite);
    2 usages
    private Player player = playerFactory.createPacMan();
    1 usage
    public Pellet pellet = new Pellet(points, sprite);
    2 usages
    private DefaultPointCalculator defaultPointCalculator = new DefaultPointCalculator();

    Thien Nguyen
    @Test
    void CollidedWithAGhostTest() { defaultPointCalculator.collidedWithAGhost(player, ghost: null); }

    Thien Nguyen
    @Test
    void consumedAPelletTest() {
        defaultPointCalculator.consumedAPellet(player, pellet);
    }
}
```

Figure 1: First

Method Paths

`src/main/java/nl/tudelft/jpacman/level/CollisionInteractionMap`

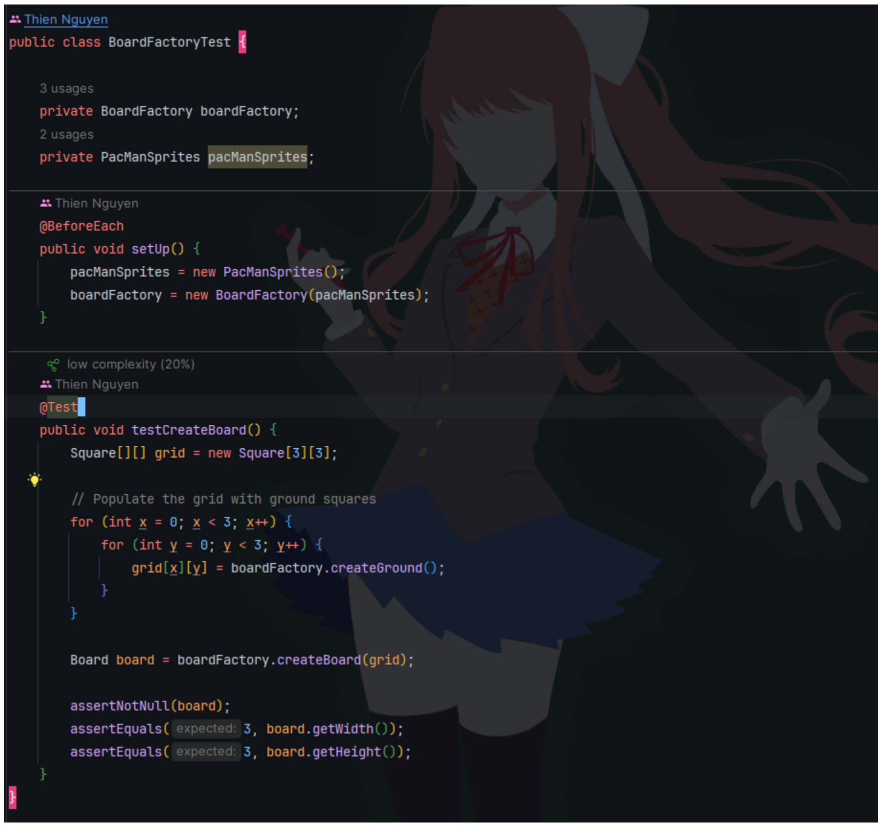


```
package nl.tudelft.jpacman.level;
import org.junit.jupiter.api.Test;

Thien Nguyen *
public class CollisionInteractionMapTest {
    no usages
    CollisionInteractionMap map = new CollisionInteractionMap();

    Thien Nguyen
    @Test
    void testOnCollision() { new CollisionInteractionMap(); }
}
```

Figure 2: First

Method Paths

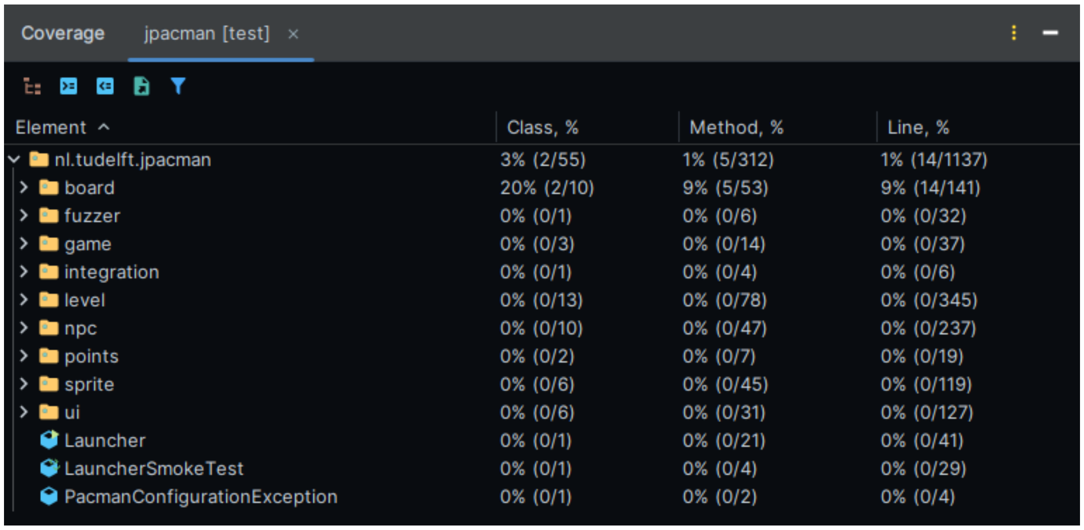
`src/main/java/nl/tudelft/jpacman/board/BoardFactory.createBoard`



```
Thien Nguyen
public class BoardFactoryTest {

    3 usages
    private BoardFactory boardFactory;
    2 usages
    private PacManSprites pacManSprites;

    Thien Nguyen
    @BeforeEach
    public void setUp() {
        pacManSprites = new PacManSprites();
        boardFactory = new BoardFactory(pacManSprites);
    }

     low complexity (20%)
    Thien Nguyen
    @Test
    public void testCreateBoard() {
        Square[][] grid = new Square[3][3];

        // Populate the grid with ground squares
        for (int x = 0; x < 3; x++) {
            for (int y = 0; y < 3; y++) {
                grid[x][y] = boardFactory.createGround();
            }
        }

        Board board = boardFactory.createBoard(grid);

        assertNotNull(board);
        assertEquals( expected: 3, board.getWidth());
        assertEquals( expected: 3, board.getHeight());
    }
}
```

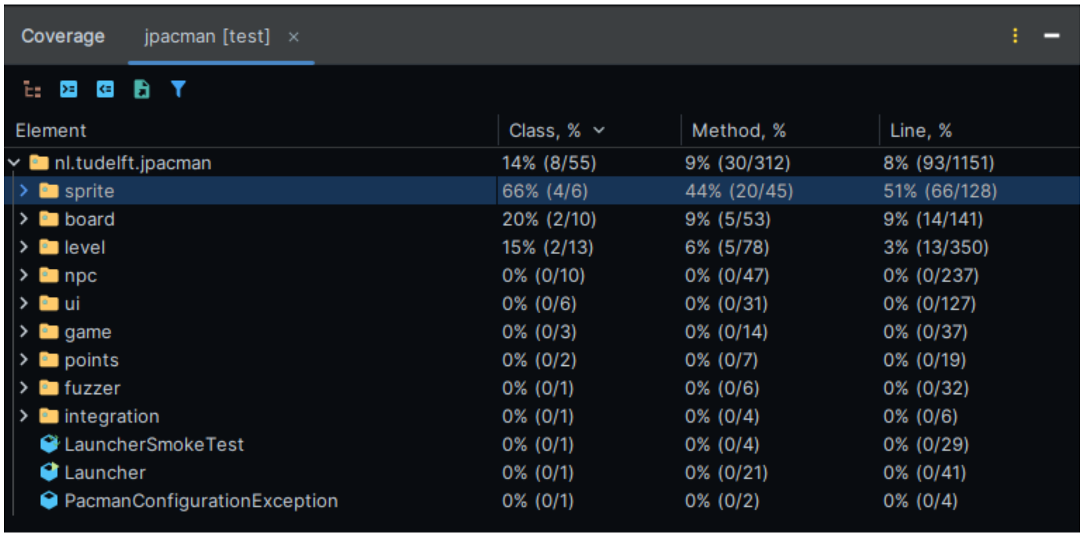
Figure 3: First

Note: Here is our report before and after the units test%.



Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	3% (2/55)	1% (5/312)	1% (14/1137)
board	20% (2/10)	9% (5/53)	9% (14/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	0% (0/13)	0% (0/78)	0% (0/345)
npc	0% (0/10)	0% (0/47)	0% (0/237)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	0% (0/6)	0% (0/45)	0% (0/119)
ui	0% (0/6)	0% (0/31)	0% (0/127)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)

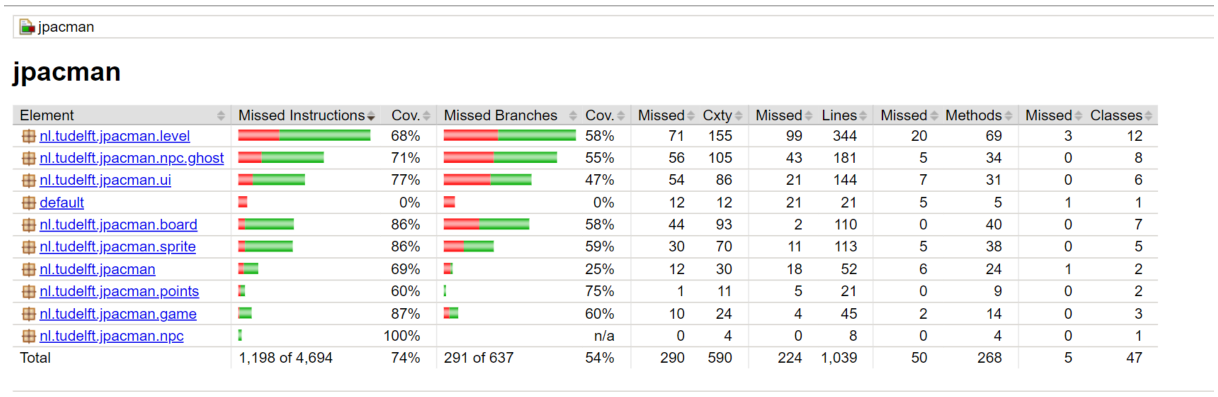
Figure 4: Before units test



Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	14% (8/55)	9% (30/312)	8% (93/1151)
sprite	66% (4/6)	44% (20/45)	51% (66/128)
board	20% (2/10)	9% (5/53)	9% (14/141)
level	15% (2/13)	6% (5/78)	3% (13/350)
npc	0% (0/10)	0% (0/47)	0% (0/237)
ui	0% (0/6)	0% (0/31)	0% (0/127)
game	0% (0/3)	0% (0/14)	0% (0/37)
points	0% (0/2)	0% (0/7)	0% (0/19)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
integration	0% (0/1)	0% (0/4)	0% (0/6)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)

Figure 5: After units test

Task 3 - JaCoCo Report on JPacman



jpacman

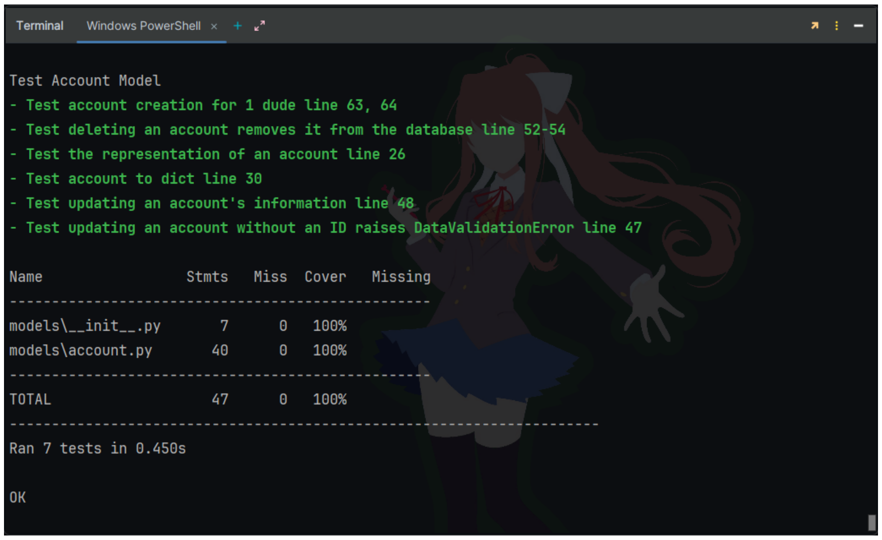
jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level		68%		58%	71	155	99	344	20	69	3	12
nl.tudelft.jpacman.npc.ghost		71%		55%	56	105	43	181	5	34	0	8
nl.tudelft.jpacman.ui		77%		47%	54	86	21	144	7	31	0	6
default		0%		0%	12	12	21	21	5	5	1	1
nl.tudelft.jpacman.board		86%		58%	44	93	2	110	0	40	0	7
nl.tudelft.jpacman.sprite		86%		59%	30	70	11	113	5	38	0	5
nl.tudelft.jpacman		69%		25%	12	30	18	52	6	24	1	2
nl.tudelft.jpacman.points		60%		75%	1	11	5	21	0	9	0	2
nl.tudelft.jpacman.game		87%		60%	10	24	4	45	2	14	0	3
nl.tudelft.jpacman.npc		100%		n/a	0	4	0	8	0	4	0	1
Total	1,198 of 4,694	74%	291 of 637	54%	290	590	224	1,039	50	268	5	47

Figure 6: JaCoCo Report on JPacman

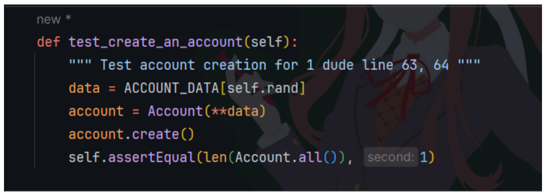
Question	**Answer**
Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why so or why not?	The coverage results from JaCoCo were **NOT** similar to those from IntelliJ. This discrepancy likely stems from the distinct focus of each tool: JaCoCo zeroes in on missed instructions and branches, providing a detailed analysis, while IntelliJ offers a broader overview by reporting on classes, methods, and lines. They simply differ in what is reported.
Did you find helpful the source code visualization from JaCoCo on uncovered branches?	Yes, the source code visualization from JaCoCo was extremely helpful. It allowed for precise identification of which branches were not covered, greatly aiding in debugging and testing.
Which visualization did you prefer and why? IntelliJ's coverage window or JaCoCo's report?	I prefer JaCoCo's report due to its detailed and specific insights, including the exact lines and branches that are not covered. However, it generates an HTML file, which is less convenient than IntelliJ's coverage window, and notably lacks a dark theme. While JaCoCo's report can be adjusted at the CSS level, this requires extra work. Depending on the project, IntelliJ's coverage window might be more useful for its convenience.

Task 4 - Working with Python Test Coverage



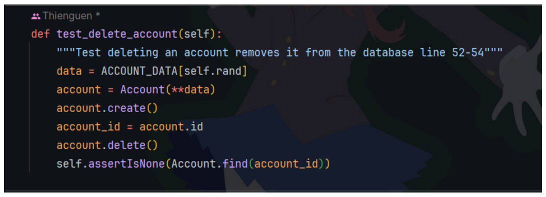
```
Test Account Model
- Test account creation for 1 dude line 63, 64
- Test deleting an account removes it from the database line 52-54
- Test the representation of an account line 26
- Test account to dict line 30
- Test updating an account's information line 48
- Test updating an account without an ID raises DataValidationError line 47

Name                 Stmts   Miss  Cover   Missing
--------------------------------------------------
models\__init__.py       7      0   100%
models\account.py       40      0   100%
--------------------------------------------------
TOTAL                   47      0   100%
----------------------------------------------------------------------
Ran 7 tests in 0.450s

OK
```

Figure 7: Python nosetests coverage report



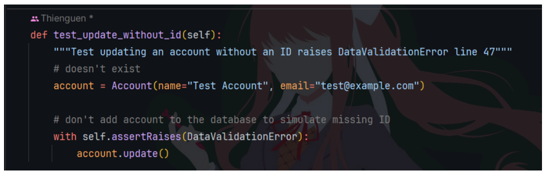
```
new *
def test_create_an_account(self):
    """ Test account creation for 1 dude line 63, 64 """
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    self.assertEqual(len(Account.all()),  second: 1)
```

(a) Python snippet 1



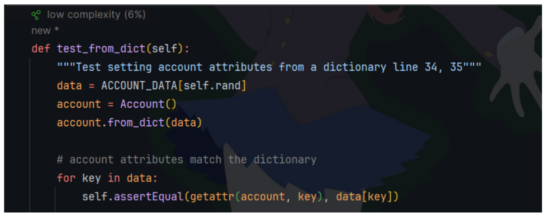
```
Thienguen *
def test_delete_account(self):
    """Test deleting an account removes it from the database line 52-54"""
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    account_id = account.id
    account.delete()
    self.assertIsNone(Account.find(account_id))
```

(b) Python snippet 3



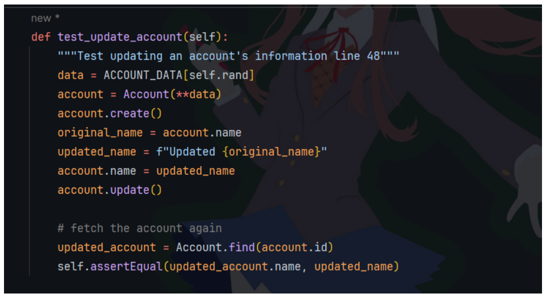
```
Thienguen *
def test_update_without_id(self):
    """Test updating an account without an ID raises DataValidationError line 47"""
    # doesn't exist
    account = Account(name="Test Account", email="test@example.com")

    # don't add account to the database to simulate missing ID
    with self.assertRaises(DataValidationError):
        account.update()
```

(c) Python snippet 2



```
low complexity (6%)
new *
def test_from_dict(self):
    """Test setting account attributes from a dictionary line 34, 35"""
    data = ACCOUNT_DATA[self.rand]
    account = Account()
    account.from_dict(data)

    # account attributes match the dictionary
    for key in data:
        self.assertEqual(getattr(account, key), data[key])
```

(d) Python snippet 5



```
new *
def test_update_account(self):
    """Test updating an account's information line 48"""
    data = ACCOUNT_DATA[self.rand]
    account = Account(**data)
    account.create()
    original_name = account.name
    updated_name = f"Updated {original_name}"
    account.name = updated_name
    account.update()

    # fetch the account again
    updated_account = Account.find(account.id)
    self.assertEqual(updated_account.name, updated_name)
```

(e) Python snippet 4

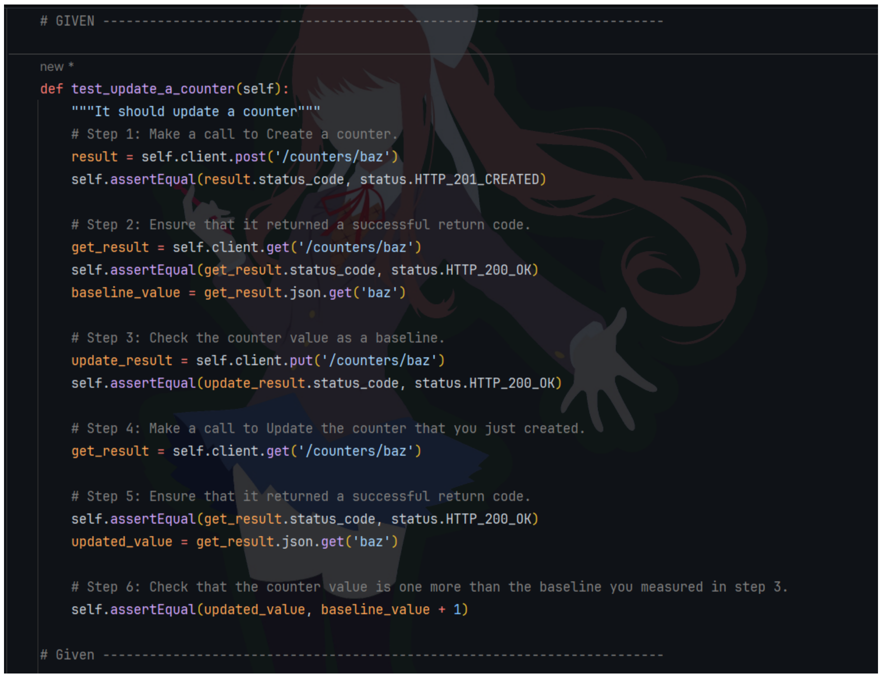
Figure 8: Python snippets collection

Task 5 - TDD

Here is my implementation of how update a counter by name in a REST API following the Test-Driven Development (TDD) methodology. The task involves writing a test case, observing it fail (Red phase), writing the minimal code to make the test pass (Green phase), and finally refactoring the code (Refactor phase).

Red Phase

The Red phase started with writing a test case named `test_update_a_counter` in `test_counter.py`. The purpose of the test was to ensure that the counter could be updated successfully using a PUT request and verify the counter's incremented value.



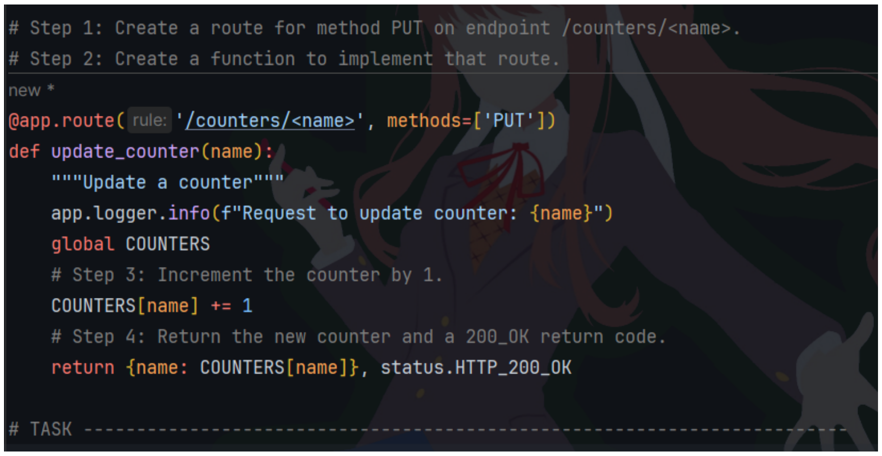
```
# GIVEN ------------------------------------------------------------------------

new *
def test_update_a_counter(self):
    """It should update a counter"""
    # Step 1: Make a call to Create a counter.
    result = self.client.post('/counters/baz')
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)

    # Step 2: Ensure that it returned a successful return code.
    get_result = self.client.get('/counters/baz')
    self.assertEqual(get_result.status_code, status.HTTP_200_OK)
    baseline_value = get_result.json.get('baz')

    # Step 3: Check the counter value as a baseline.
    update_result = self.client.put('/counters/baz')
    self.assertEqual(update_result.status_code, status.HTTP_200_OK)

    # Step 4: Make a call to Update the counter that you just created.
    get_result = self.client.get('/counters/baz')

    # Step 5: Ensure that it returned a successful return code.
    self.assertEqual(get_result.status_code, status.HTTP_200_OK)
    updated_value = get_result.json.get('baz')

    # Step 6: Check that the counter value is one more than the baseline you measured in step 3.
    self.assertEqual(updated_value, baseline_value + 1)

# Given ------------------------------------------------------------------------
```

Figure 9: Python TDD nosetests 1

The test initially failed due to a 405 Method Not Allowed error, indicating that the PUT route for updating the counter was not implemented.

0.1 Green Phase

To resolve the failing test and enter the Green phase, the following code was added to `counter.py`:



```
# Step 1: Create a route for method PUT on endpoint /counters/<name>.
# Step 2: Create a function to implement that route.
new *
@app.route( rule: '/counters/<name>', methods=['PUT'])
def update_counter(name):
    """Update a counter"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    # Step 3: Increment the counter by 1.
    COUNTERS[name] += 1
    # Step 4: Return the new counter and a 200_OK return code.
    return {name: COUNTERS[name]}, status.HTTP_200_OK

# TASK ----------------------------------------------------------------------
```

Figure 10: Python TDD nosetests 2

This code snippet successfully addressed the failing test by implementing the PUT route and incrementing the counter by 1.

0.2 Refactor Phase

We added the code mentioned in the counter.py to define the router, hence fixed the errors