LOG6305 - Techniques avancées de test du logiciel

Assignment 1

Unit tests with Pytest

Département de génie informatique et de génie logiciel

École Polytechnique de Montréal

Hiver 2024

1 Introduction

The creation of unit tests is one of the important steps in the software development. The goal of unit testing is to validate that each unit of a software under test is working as intended. The unit is the smallest testable component of a software. Typically, this component has one or more inputs and only one output.

In this lab, you will generate unit tests for Python software. Python is popular in many fields such as data science and machine learning, and is now one of the most widely used programming languages (see for example the IEEE ranking Spectrum [1]). In addition, Python is the dynamically typed programming language. Automatic test generation for dynamically typed languages is an important research problem [2].

The most common frameworks for Python testing are unittest [3] and Pytest [4]. In our course, we're going to use the Pytest framework.

In this lab, you will master the basic concepts of creating unit tests in Python with Pytest. You'll evaluate your tests according to the coverage criteria based on the control graph structural analysis and data flow analysis.

2 Objectives

The objectives of this work are: :

1. 1. Learn how to use Pytest to manage unit tests in Python
2. Learn the concept of black box and white box testing.
3. Practice designing test sets satisfying various coverage criteria such as code coverage criteria based on control flow and data flow graph analysis as well as condition coverage criteria.

3 Pytest basics

Function-based testing

In Pytest, each test is a simple Python function. Test functions must be named with the prefix `test_` to be automatically detected. Pytest can automatically detect test files and test functions in a project. Running the tests is done simply by running the command `pytest` in the project directory. Pytest uses assertions to verify that the behavior of the code under test conforms to expectations. In the next paragraph, we give a basic example of using Pytest.

[1] `https://spectrum.ieee.org/the-top-programming-languages-2023`
[2] `https://arxiv.org/abs/2111.05003`
[3] `https://docs.python.org/3/library/unittest.html`
[4] `https://docs.pytest.org/en/7.4.x/`

Usage example

Create a python module named `calculator.py` with the following function :

```
# calculator.py
def add(a, b):
    return a + b
```

Create a test file named `test_calculator.py` with the following test for the addition function :

```
# test_calculator.py
from calculator import add
def test_addition():
    result = add(2, 3)
    assert result == 5
```

se Pytest to run the tests. Make sure Pytest is installed by running `pip install pytest`. Run the tests with the following command:

```
pytest test_calculator.py
```

Fixtures

Fixtures are special Python functions that can be used to configure a test environment before running one or more tests. They help avoid code redundancy by providing a way to share configurations across multiple tests. You can see the following example where we use the 'input values' fixture to configure the input data.

```
# test_calculator.py
import pytest
from calculator import add

# Fixture pour initialiser les valeurs
@pytest.fixture
def input_values():
    a = 2
    b = 3
    return a, b

def test_addition(input_values):
    a, b = input_values
    result = add(a, b)
    assert result == 5
```

Fixtures can be used to initialize objects, configure database connections, or perform other configuration tasks necessary for testing. Testing should not be limited to a single fixture. Tests can depend on as many fixtures as needed, and fixtures can also use other fixtures.

Test parametrization :

Pytest allows to parimetrize the tests using the decorator `@pytest.mark.parametrize` . This allows the same test to be run with different input data. Here is an example, where the @parametrize decorator defines three different tuples '(input_a, input_b, expected)' so that the 'test_addition' function executes three times using them in turn:

```
import pytest
@pytest.mark.parametrize("input_a, input_b, expected", [
    (2, 3, 5),
    (0, 5, 5),
    (-2, -2, -4),
])
def test_addition(input_a, input_b, expected):
    result = add(input_a, input_b)
    assert result == expected
```

The decorator `@pytest.mark.parametrize` allows you to write more compact tests and execute them with different inputs.

In conclusion, using Pytest simplifies the process of writing and running unit tests in Python. To go further and explore all the features of Pytest, we strongly encourage you to consult the official Pytest documentation[5].

4 Black box testing

In black box testing, test data is selected either randomly or from the functional specifications of the program. The internal structure (code) of the program is therefore unknown or ignored. The tester considers the program under test as a black box where only the inputs, outputs and function of the program are known. There are several so-called 'black box' testing techniques. Below we discuss some of the commonly used ones [1].

The technique of categories and partitions. The category and partition technique [2] assumes a partitioning of the input domains into equivalence classes. It is based on the following intuition: each test data point is representative of its equivalence class in the ability to reveal faults, i.e., the data points from the same class have the same probability of revealing a failure.

The boundary values technique.The boundary value technique is a variation of the previous technique by considering not only the values inside the equivalence classes, but also at the limits of each class.

5 White box testing

Unlike black box testing, in white box testing the test data is selected with the aim of satisfying criteria related to the code structure. The code under test and its properties

[5]https://docs.pytest.org/

(control and data flow graphs) are available to the tester, and the tests can be designed according to the implementation details of the program.

Criteria based on control flow analysis. The criteria most used in an industrial environment are the instruction coverage criterion and the branch coverage criterion. These criteria measure respectively the percentage of instructions and branches of the program that have been executed at least once during the test [1]. A branch in a control flow graph corresponds to one of the two possible evaluations of the decision that a node of the graph contains.

Instruction coverage. A basic requirement of this criterion is that all program instructions are executed by the tests. Although execution of every instruction is required, the instruction coverage is a weak criterion. Indeed, a set of tests that covers all instructions does not necessarily ensure coverage of certain transfers of control in the program.

Branch coverage. Branch coverage takes into consideration the fact that branching statements (if s, for s, while s, etc.) cause the program to behave in different ways, depending on how the statement is evaluated. For a simple if(a && b) statement, having a test case T1 that makes the if statement true and another test case T2 that makes the statement false is enough to consider the branch covered.

Condition coverage criteria. Based on a decomposition of a decision (branch) into atomic conditions, called predicates or clauses [1] other criteria refining the branch coverage criterion have been proposed. In this laboratory we will use the MC/DC criterion which is one of the most used criteria in industrial applications. In particular, in critical applications in fields such as aviation. Here is the formal definition of MC/DC [1] *A set of execution paths P satisfies the MC/DC coverage criterion if and only if P satisfies the coverage of all instructions, and for each decision, all possible combinations of conditions whose variation independently influences the value of the decision have been tested at least once.*

Criteria based on data flow analysis. In these criteria, coverage is defined based on the analysis of subpaths linking the definitions of variables to their uses. This analysis highlights how values are associated with their definitions and how these associations can affect program execution, and focuses on the occurrences of variables in the program. Any occurrence of a variable is considered either a definition (the variable is linked to a value) or a use (the variable is referenced) [[1]. The following coverage criteria are commonly used:

All definitions coverage (all-defs). Test cases must cover a path in which a definition achieves the use of that definition.

All uses coverage. Test cases must cover a path in which a definition reaches all uses of that definition.

All du paths coverage. Test cases must cover all the paths from the variable definition to all of its uses.

Example. Let's imagine that we have obtained a data flow graph for our program as shown in Figure 1. To achieve the All-uses coverage, we need to cover the following paths:

- d1(x) to u2(x)
- d1(x) to u3(x)
- d1(x) to u5(x)

The paths satisfying the All-uses criterion are:

- {1, 2, 4, 5, 6}
- {1, 3, 4, 6}

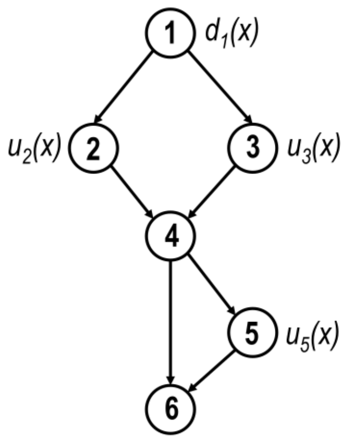

Figure 1: Data flow graph example

6 The tasks

Required set-up

Ensure the **Python** and **PyCharm** [6] (you have access to a free license as a student) or VSCode [7] are installed.
Operational system : Linux/macOS/Windows **Python** : version 3.10

Obtain a local copy of the code for the first practical work, that is named 'count_words.py' This code implements a program that has the following requirements:

> *Given a sentence, the program should count the number of words that end with either 's' or 'r'. A word ends when a non-letter appears. The program returns the number of words.*

Your tasks:

Your main task is to test this program and ensure that it meets the requirements.

1. Question 1: Write black-box tests with Pytest for this program according to the requirements specified. Use partition and boundary analysis techniques. Save your solution in the 'test_q1.py' file. Explain how you made the partitions and what boundary values you used (3 points).

[6]`https://www.jetbrains.com/help/pycharm/pycharm-educational.html`
[7]`https://code.visualstudio.com/`

2. Question 2: Check if the tests you generated in Question 1 satisfy the instruction coverage criterion (100% line coverage must be obtained). If yes, choose the minimum test quantity that satisfy the line coverage and save your solution in the ‘test_q2.py’ file. Otherwise, write additional tests to get 100% line coverage and save your solution in the ‘test_q2.py’ file (1 points).

3. Question 3: Check if the tests you generated in Question 1 satisfy the branch coverage criterion (100% branch coverage must be obtained). If yes, choose the minimum test quantity that satisfy the branch coverage and save your solution in the ’test q3.py’ file. Otherwise, write additional tests to achieve 100% branch coverage. Save your solution in the ’test q3.py’ file. In the laboratory report, explain how each branch is covered with your tests (3 points).

4. Question 4: Generate test cases using Pytest that satisfy the criterion of covering MC/DC conditions. Save your solution in the ’test q4.py’ file. In the laboratory report, explain how your tests meet the criterion, i.e. why you chose the inputs shown in your tests (4 points).

5. Question 5: Draw a graph of the data flow of the program ’count words.py’ and add annotations for the DEFs and USEs of the variables ’last’ and ’words’ as in 1. Indicate which paths should be covered to obtain ALL-DEFs, ALL-USEs and ALL-DU paths coverage (3 points)

6. Question 6: Generate test cases using Pytest that satisfy the criterion of data flow coverage such as ALL-DEFs, ALL-USEs and ALL-DU paths for the variable ’last’ and ’words ’. Save your solution in the ’test_q6.py’ file. In the laboratory report, explain how your tests meet the criterion, i.e. show which path is exercised by each test (4 points).

7. Question 7: Please analyze your results: how many tests have you created to achieve the following coverage criteria: line coverage, branch coverage, MC/DC coverage, ALL-DU paths coverage? (0.5 points) What is the difference between branch coverage and condition coverage? (0.5 points) What is the difference between the MC/CD criterion and other conditions coverage criteria? (0.5 points) Which criterion is the most useful and practical in your opinion? (0.5 points).

Important instructions:

1. In your tests in questions 1, 2, 3, 4 and 6 you should use at least 1 fixture and 1 parameterization (for each question) (-1 per question, if fixture or parameterization are not used).

2. Test case names should be descriptive (the test name should help us understand what it is testing). In addition, for each test add a short comment with the description of the test and its function (-0.2 for each test without comment).

7 Expected deliverables

The following deliverables are expected:

- The report for this practical work in the PDF format. The report should contain answers to questions 1, 3-7.
- The folders 'test_q1.py', 'test_q2.py' and 'test_q3.py', 'test_q4.py', 'test_q6.py' that contain your tests.

You should submit all the files in a single **zip** archive with the following name student_id_lab1.zip on Moodle. Additionally, you should add to the .zip a ".txt" file with the following content: "Your full name, your student id" (this might be used for the grading automation). This assignment is individual.

The report file must contain the title and number of the laboratory, your name and student id.

8 Important information

1. Check the course Moodle site for the file submission deadline
2. A delay of [0.24 hours] will be penalized by 10%, [24 hours, 48 hours] by 20% and more than 48 hours by 50%.
3. No plagiarism is tolerated (including ChatGPT). You should only submit code created by you.

References

[1] Abdesselam Lakehal. *Critères de couverture structurelle pour les programmes Lustre.* PhD thesis, Université Joseph-Fourier-Grenoble I, 2006.

[2] John B Goodenough and Susan L Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, 1975.