TESTING
Introduction

RSE Summer School, September 24, 2024 | Jakob Fritz, Maria Lupe Barrios Sazo, Dirk Brömmel, Robert Speck | Jülich Supercomputing Center, Forschungszentrum Jülich

Overview

Schedule I

Talk

- Why to do testing
- What is “coverage”
- How to differentiate tests (Two dimensions)
- Different scopes of tests
- Different types of tests
- Popular testing frameworks
- When to use what test
- Examples

Getting things done

Coffee break

Overview

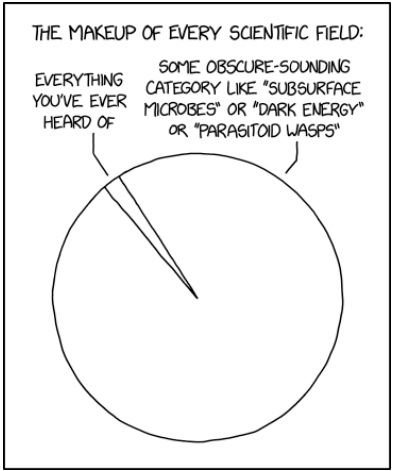
Schedule II

Talk

- How to differentiate tests (extension)
- More different scopes of tests
- More different types of tests
- Examples

Getting more things done

Overview



Comic from XKCD (2986).

We all don’t know everything:

- Regardless of how much you already know about testing, there is stuff about testing, that you don’t know yet.
- This talk shall give ideas where and how to start or improve your test suit

Overview

Why to do testing

Reason for testing:
⇒ Finding bugs
Reason for finding bugs:
⇒ Making the user happy (generally) / making the results reproducible (in science)

Based on Wacker 2015

Overview

Why to do testing

Reason for testing:
⇒ Finding bugs
Reason for finding bugs:
⇒ Making the user happy (generally) / making the results reproducible (in science)

So what makes a user happy / the results reproducible?

Based on Wacker 2015

Overview

Why to do testing

Reason for testing:
⇒ Finding bugs
Reason for finding bugs:
⇒ Making the user happy (generally) / making the results reproducible (in science)

So what makes a user happy / the results reproducible?

Test added → Test fails → Bug reported → Bug fixed




Based on Wacker 2015

Overview

Coverage

What part of the code is covered?

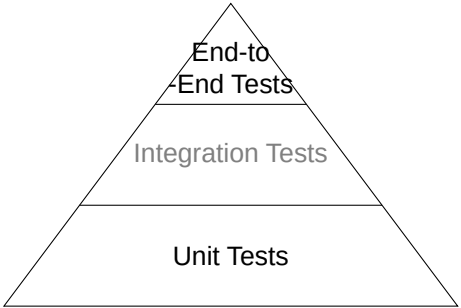
Different metrics for coverage:

- **Line coverage $\Leftarrow$ Which line was (partly) executed; Most often used**
- Path coverage $\Leftarrow$ Which path (conditionals, …) was executed
- Statement coverage $\Leftarrow$ Which statements (even within a line) were executed

Overview

Two dimensions

Two individual dimensions to distinguish tests by scope and type

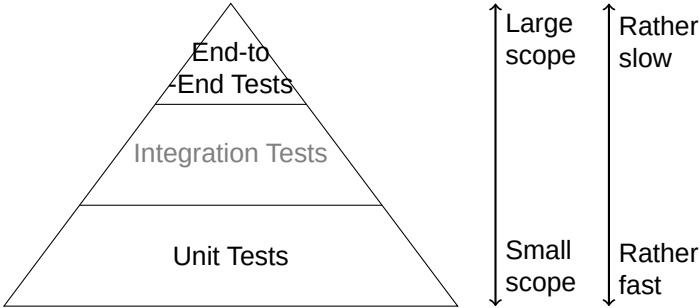


Distinction by scope and proposed fractions are based on Wacker 2015

Overview

Two dimensions

Two individual dimensions to distinguish tests by scope and type

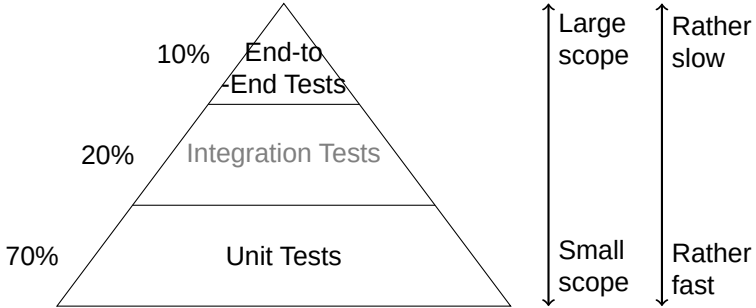


Distinction by scope and proposed fractions are based on Wacker 2015

Overview

Two dimensions

Two individual dimensions to distinguish tests by scope and type

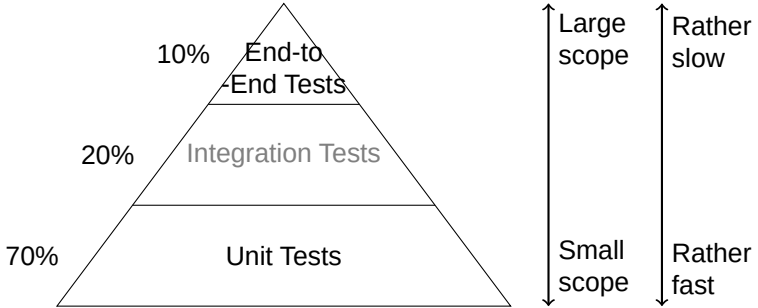


Distinction by scope and proposed fractions are based on Wacker 2015

Overview

Two dimensions

Two individual dimensions to distinguish tests by scope and type



Type of the test

- Smoke test
- Golden master test
- Property based test
- Fuzzy test
- Mutation test

Distinction by scope and proposed fractions are based on Wacker 2015

Distinguish by scope

Unit tests

Testing the smallest building blocks of your code. Often these are functions.
Ideally hermetic[1] tests with little or no side effects.

[1] "Test in a box", no external dependencies

Distinguish by scope

Unit tests

Testing the smallest building blocks of your code. Often these are functions.
Ideally hermetic[1] tests with little or no side effects.

Pros:

- Rather fast to execute
- Helpful to locate bugs, as little code is tested per test
- Easy to write if code is organized in many small functions
- Hermetic tests have fewer sources of flakiness

Cons:

- Hard to write if functions are highly integrated
- Does not check behavior of the complete codebase/system

[1] "Test in a box", no external dependencies

Distinguish by scope

End-to-End tests

Testing the entire codebase. I.e. a complete pipeline. This could be reading in data, processing it, and storing all results in various formats (table, figures, websites, …).

Distinguish by scope

End-to-End tests

Testing the entire codebase. I.e. a complete pipeline. This could be reading in data, processing it, and storing all results in various formats (table, figures, websites, …).

Pros:

- Tests more realistic usage of the codebase
- Easy to implement even for strongly integrated code

Cons:

- Rather slow to execute
- Can be prone to errors or external issues (e.g. flaky internet connection, or lacking availability of a service)
- Large codebase tested in single job, so hard to track origin of bugs

Distinguish by type

Smoke tests

Distinguish by type

Smoke tests

Electrical devices run on magic smoke.
Once the smoke leaves the device, it stops working.

Smoke tests often are easy to implement, as they only tests for failures.
There are no particular checks. Just compile and run your code. If errors occur, this test failed.

Distinguish by type

Smoke tests

Electrical devices run on magic smoke.
Once the smoke leaves the device, it stops working.

Smoke tests often are easy to implement, as they only tests for failures.
There are no particular checks. Just compile and run your code. If errors occur, this test failed.

Pros:

- Easy to implement
- No need to update when changing the code

Cons:

- Not specific (what error from where)
- Does not check if results are as expected
- May take long (End-2-End test)

Distinguish by type

Golden master tests

Check if specific examples still work as expected.

Specify input-data and expected output. Calculated output from input-data and compare with expected output.

Distinguish by type

Golden master tests

Check if specific examples still work as expected.

Specify input-data and expected output. Calculated output from input-data and compare with expected output.

Pros:

- Well suited for examples (e.g. from the Tutorials / Docs)
- Can detect changes of computational results (though not always)
- Comparably easy to test cases with complex input- or output-data
- Comparably easy to understand & implement

Cons:

- Limited test scope (as only a few input-output-combinations are tested)

Popular Frameworks

	Single case	Mocking	Property based testing	Mutation testing
C++	Catch2 & GoogleTest	GoogleTest	rapidcheck	
Rust	Cargo tests (builtin)	Mockall	Quickcheck & proptest	Cargo-mutants
Python	Pytest & Unittest	Mock	Hypothesis	Mutatest
Java	JUnit	Mockito	jqwik & junit-quickcheck	Pitest
Julia	Unit Testing	Mocking		

When to use what

Phase	Input	Processing	Output
Examples	Read file Query (measurement) device Query API …	Process your data This is your ‘actual work’ Your magic goes here …	Store the results In graphs, text-files, tables, … Push to a database …
Testing	Often contains parsing and interaction with external resources (use supplied functions) Put emphasis on parsing/validating the input, not on reading the files (split into separate units)	Put most emphasis on testing here, as this is often the most difficult work	Testing is more difficult, but components are more standard (use supplied functions)

When to use what

Scope

Modular code:

```
/
├── setup.py
├── calculator
│   ├── __init__.py
│   ├── file_interaction.py
│   ├── main.py
│   └── perform_calculations.py
└── testing
    ├── test_e2e.py
    ├── test_main.py
    └── test_perform_calculations.py
```

- Unit tests
- A single or few End-to-End tests

Highly integrated code:

```
/
├── setup.py
├── calculator
│   ├── __init__.py
│   └── main.py
└── testing
    └── test_e2e.py
```

- End-to-End tests

When to use what

Type

Regardless of how integrated the code is:
Implement tests of a type to a high coverage before starting with the next type.

When you already finished implementing your code

1. Smoke Tests (for End-to-End tests)
2. Golden master tests
3. Property based testing (for unit tests)

While you still implement your code

1. Golden master tests (for unit tests)
2. Smoke Tests (for End-to-End tests)
3. Golden master tests (for End-to-End tests)
4. Property based testing (for unit tests)

Examples

```
/
├── setup.py
├── calculator
│   ├── __init__.py
│   ├── file_interaction.py
│   ├── main.py
│   └── perform_calculations.py
└── testing
    ├── test_e2e.py
    ├── test_main.py
    ├── test_perform_calculations.py
    └── test_perform_calculations_gm.py
```

Files from: https://jugit.fz-juelich.de/rg-rse/testing-calculator

Examples

file_interaction.py

```
6  def read_file(filename):
7      with open(filename, "r") as f:
8          equation_strings = f.readlines()
9      return equation_strings
10
11
12 def write_file(filename, content):
13     with open(filename, "w") as f:
14         f.writelines("\n".join(content))
```

Examples

main.py

```
10  def split_equation(eq_string):
11      pattern = r"^\s*(-?[\d\.]+)\s*([\+\-\*\/%])\s*(-?[\d\.]+)\s*$"
12      m = re.match(pattern, eq_string)
13      if m:
14          a, operator, b = (m.group(1), m.group(2), m.group(3))
15      else:
16          raise ValueError("...")
25      return (a, b, operator)
```

Examples

main.py

```
28  def combine_results(a, b, c, op):
29      return f"{a} {op} {b} = {c}"
30
31
32  def main():
33      eq_string_list = file_interaction.read_file("input.txt")
34      result_list = []
35      for eq_string in eq_string_list:
36          if eq_string.strip() == "":
37              result_list.append("")
38          a, b, op = split_equation(eq_string)
39          c = perform_calculations.solve_calculation(a, b, op)
40          result_list.append(combine_results(a, b, c, op))
41      file_interaction.write_file("results.txt", result_list)
```

Examples

perform_calculations.py

```
26  def solve_calculation(a, b, op):
27      match op:
28          case "+":
29              c = run_addition(a, b)
30          case "-":
31              c = run_subtraction(a, b)
32          case "*":
33              c = run_multiplication(a, b)
34          case "/":
35              c = run_division(a, b)
36          case "%":
37              c = run_modulo(a, b)
38          case _:
39              raise ValueError("...")
40      return c
```

Examples

How to test them

```
/
├── setup.py
├── calculator
│   ├── __init__.py
│   ├── file_interaction.py
│   ├── main.py
│   └── perform_calculations.py
└── testing
    ├── test_e2e.py
    ├── test_main.py
    ├── test_perform_calculations.py
    └── test_perform_calculations_gm.py
```

Examples

test_e2e.py

A smoke test as end-to-end test

```
5  import os
6  from calculator import main
7
8
9  def test_main():
10     main.main()
11     # Assert result-file exists
12     assert os.path.isfile("results.txt")
13     # Assert they have the same number of lines
14     with open("input.txt", "rb") as f:
15         num_lines_input = sum(1 for _ in f)
16     with open("results.txt", "rb") as f:
17         num_lines_results = sum(1 for _ in f)
18     assert num_lines_input == num_lines_results
```

Examples

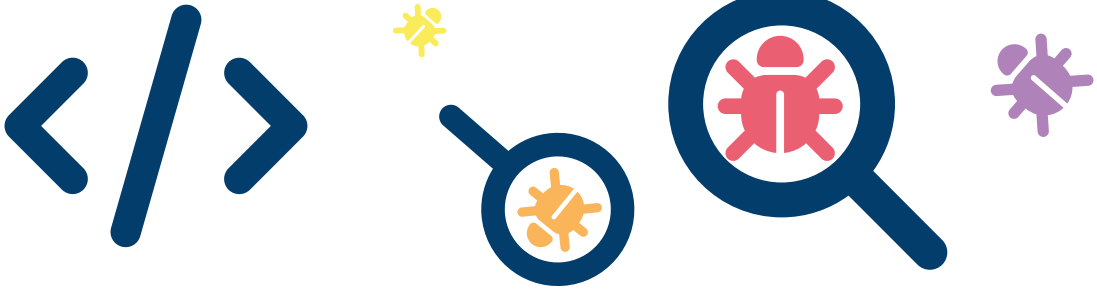
test_perform_calculations_gm.py

A golden master test for calculations

```
5  import pytest
6  from calculator import perform_calculations
7
8
9  @pytest.mark.parametrize("a, b, res", [(1, 1, 2), (2, 3, 5), (-2, 1, -
10 def test_run_addition(a, b, res):
11     result = perform_calculations.run_addition(a, b)
12     assert result == res
13
14
15 @pytest.mark.parametrize("a, b, res", [(1, 1, 1), (0, 2, 0), (-2, 1, -
16 def test_run_multiplication(a, b, res):
17     result = perform_calculations.run_multiplication(a, b)
18     assert result == res
```

Getting things done

Now, enough talking.
You can use the time until the break to write first tests for your code.



TESTING
Advanced

RSE Summer School, September 24, 2024 | Jakob Fritz, Maria Lupe Barrios Sazo, Dirk Brömmel, Robert Speck | Jülich Supercomputing Center, Forschungszentrum Jülich

Overview

Schedule II

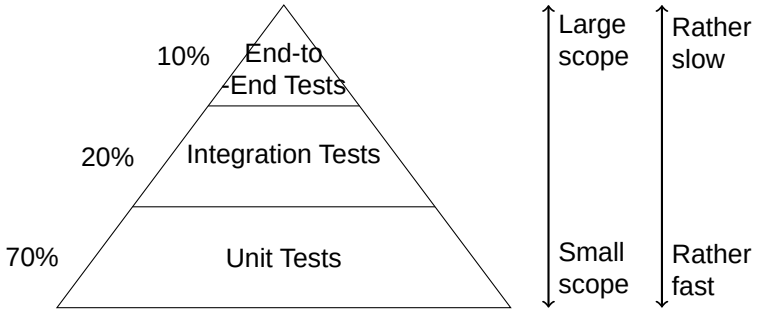
Talk

- How to differentiate tests (extension)
- More different scopes of tests
- More different types of tests
- Examples

Getting more things done

Overview

Two individual dimensions to distinguish tests by scope and type



Type of the test

- Smoke test
- Golden master test
- Property based test
- Fuzzy test
- Mutation test

Distinction by scope and proposed fractions are based on Wacker 2015

Distinguish by scope

Integration tests

Knowing your units work is good, but it is also relevant to know if they interact properly.
Often this means checking interaction of only a few units (most often only two).

Distinguish by scope

Integration tests

Knowing your units work is good, but it is also relevant to know if they interact properly.
Often this means checking interaction of only a few units (most often only two).

Pros:

- Check interaction between two or more units
- Faster than End-to-End tests
- Realize early if parameters are handled incorrectly

Cons:

- Can be hard to write depending on structure of code
- Slower than unit tests
- Less specific on where errors came from (if not combined with unit tests)

Distinguish by type

Property based tests

Property based testing does not check for specific values, but checks properties of variables. This could be types, sizes (of lists, matrices, …), or if lists are ordered. Furthermore, the input-data is created by following constraints. This may uncover bugs not thought of.

Further reading: Hypothesis blog

Distinguish by type

Property based tests

Property based testing does not check for specific values, but checks properties of variables. This could be types, sizes (of lists, matrices, …), or if lists are ordered. Furthermore, the input-data is created by following constraints. This may uncover bugs not thought of.

Examples when working with text include long texts, texts with line-breaks, non-alphabetical symbols such as commas, slashes, brackets, or Unicode in general.

Further reading: Hypothesis blog

Distinguish by type

Property based tests

Property based testing does not check for specific values, but checks properties of variables. This could be types, sizes (of lists, matrices, …), or if lists are ordered. Furthermore, the input-data is created by following constraints. This may uncover bugs not thought of.

Examples when working with text include long texts, texts with line-breaks, non-alphabetical symbols such as commas, slashes, brackets, or Unicode in general.

Common issues when working with floats are very large values (issues with overflows), infinity and NAN, as well as values very near to zero.

Further reading: Hypothesis blog

Distinguish by type

Property based tests

Property based testing does not check for specific values, but checks properties of variables. This could be types, sizes (of lists, matrices, …), or if lists are ordered. Furthermore, the input-data is created by following constraints. This may uncover bugs not thought of.

Pros:

- Helps to find bugs for uncommon values of inputs

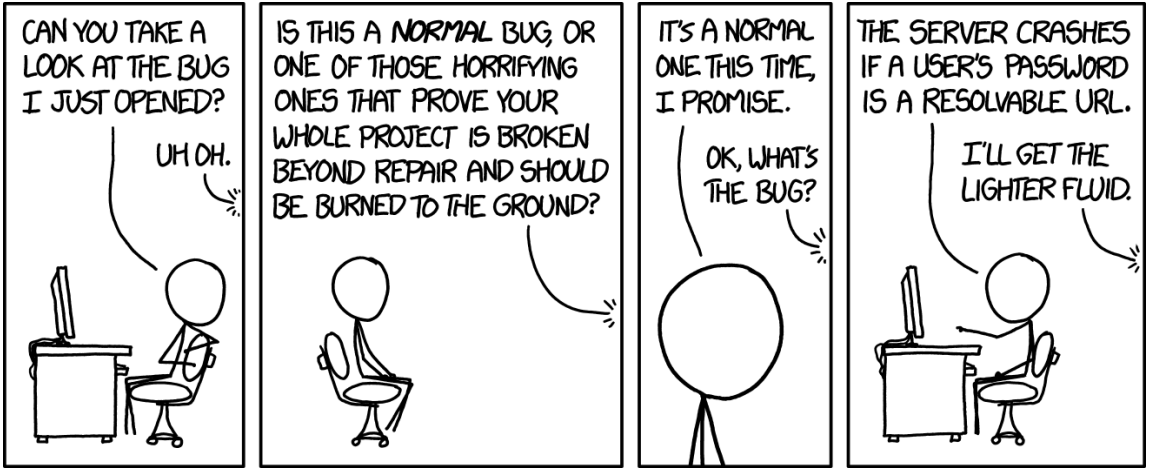
Cons:

- Difficult to create complex data structures
- An addition rather than replacement for golden master tests

Further reading: Hypothesis blog

Distinguish by type

Property based tests



Comic from XKCD (1700).

JÜLICH Forschungszentrum

Distinguish by type

Fuzzy tests

Fuzzy tests are similar to property based tests, but the used inputs are more general and the results are checked less precise.

Distinguish by type

Fuzzy tests

Fuzzy tests are similar to property based tests, but the used inputs are more general and the results are checked less precise.

Pros:

- Testing functions for robustness against user- or interaction errors
- Finding more edge-cases that raise errors than property based testing

Cons:

- Rather a smoke test for a wide range of inputs
- No test for correctness, but for errors

Distinguish by type

Mutation Tests

In mutation tests, the code is changed, and the test suite runs again. By this, it is checked if the tests are sensitive enough to detect the changes.
⇒ This is not about testing your code but about testing your test suite.

Quote by Evan Kepner 2020

Distinguish by type

Mutation Tests

In mutation tests, the code is changed, and the test suite runs again. By this, it is checked if the tests are sensitive enough to detect the changes.
⇒ This is not about testing your code but about testing your test suite.

> *Essentially, mutation testing is a test of the alarm system created by the unit tests.*

Quote by Evan Kepner 2020

Distinguish by type

Mutation Tests

In mutation tests, the code is changed, and the test suite runs again. By this, it is checked if the tests are sensitive enough to detect the changes.
⇒ This is not about testing your code but about testing your test suite.

> *Essentially, mutation testing is a test of the alarm system created by the unit tests.*

Pros:

- ➕ Testing the actual coverage of your tests (not only lines)

Cons:

- ➖ Does not improve your code (-coverage) but hints where to improve your tests

---

Quote by Evan Kepner 2020

Examples

```
/
├── setup.py
├── calculator
│   ├── __init__.py
│   ├── file_interaction.py
│   ├── main.py
│   └── perform_calculations.py
└── testing
    ├── test_e2e.py
    ├── test_main.py
    ├── test_perform_calculations.py
    └── test_perform_calculations_gm.py
```

Examples

file_interaction.py

```
6  def read_file(filename):
7      with open(filename, "r") as f:
8          equation_strings = f.readlines()
9      return equation_strings
10
11
12 def write_file(filename, content):
13     with open(filename, "w") as f:
14         f.writelines("\n".join(content))
```

Examples

main.py

```
10  def split_equation(eq_string):
11      pattern = r"^\s*(-?[\d\.]+)\s*([\+\-\*\/%])\s*(-?[\d\.]+)\s*$"
12      m = re.match(pattern, eq_string)
13      if m:
14          a, operator, b = (m.group(1), m.group(2), m.group(3))
15      else:
16          raise ValueError("...")
25      return (a, b, operator)
```

Examples

main.py

```
28  def combine_results(a, b, c, op):
29      return f"{a} {op} {b} = {c}"
30
31
32  def main():
33      eq_string_list = file_interaction.read_file("input.txt")
34      result_list = []
35      for eq_string in eq_string_list:
36          if eq_string.strip() == "":
37              result_list.append("")
38          a, b, op = split_equation(eq_string)
39          c = perform_calculations.solve_calculation(a, b, op)
40          result_list.append(combine_results(a, b, c, op))
41      file_interaction.write_file("results.txt", result_list)
```

Examples

perform_calculations.py

```
26  def solve_calculation(a, b, op):
27      match op:
28          case "+":
29              c = run_addition(a, b)
30          case "-":
31              c = run_subtraction(a, b)
32          case "*":
33              c = run_multiplication(a, b)
34          case "/":
35              c = run_division(a, b)
36          case "%":
37              c = run_modulo(a, b)
38          case _:
39              raise ValueError("...")
40      return c
```

Examples

How to test them

```
/
├── setup.py
├── calculator
│   ├── __init__.py
│   ├── file_interaction.py
│   ├── main.py
│   └── perform_calculations.py
└── testing
    ├── test_e2e.py
    ├── test_main.py
    ├── test_perform_calculations.py
    └── test_perform_calculations_gm.py
```

Examples

test_main.py

A property based test on parsing text

```
29  @given(
30      a=st.integers(),
31      b=st.integers(),
32      op=st.sampled_from(["+", "-", "*", "/", "%"]),
33      ws1=st.text(alphabet=st.characters(categories=["Zs"], include_char
34      ws2=st.text(alphabet=st.characters(categories=["Zs"], include_char
35      ws3=st.text(alphabet=st.characters(categories=["Zs"], include_char
36      ws4=st.text(alphabet=st.characters(categories=["Zs"], include_char
37  )
38  def test_split_equation_whitespace(a, b, op, ws1, ws2, ws3, ws4):
39      string1 = f"{ws1}{a}{ws2}{op}{ws3}{b}{ws4}"
40      ra, rb, rop = main.split_equation(string1)
41      assert (ra, rb, rop) == (a, b, op)
```

Examples

test_perform_calculations.py

A property based test for calculations

```
11  @given(a=st.integers(), b=st.integers())
12  @example(a=3, b=2)
13  def test_run_addition(a, b):
14      result = perform_calculations.run_addition(a, b)
15      result_2 = perform_calculations.run_addition(b, a)
16      assert result == result_2
17      if b > 0:
18          assert result > a
19      elif b < 0:
20          assert result < a
21      else:  # b==0
22          assert result == a
23      if (a, b) == (2, 3):
24          assert result == 5
```

Getting things done

Now, enough talking.
You can use the time until the end to extend your tests and to increase test coverage for your code.

Wrap up

Thank you for taking time to work on your code and on

Wrap up

Thank you for taking time to work on your code and on

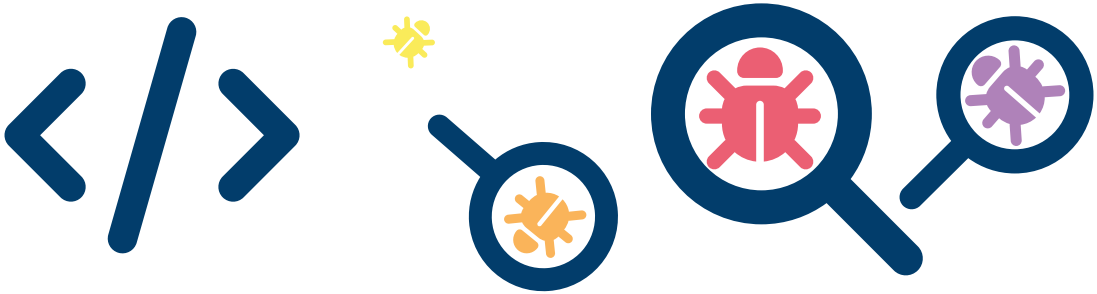
What did you like the most or what was the most interesting to you?

Wrap up

Thank you for taking time to work on your code and on

What did you like the most or what was the most interesting to you?

Feel free to approach the other tutors or me the whole week if questions arise!



TESTING
Goodbye

RSE Summer School, September 24, 2024 | Jakob Fritz, Maria Lupe Barrios Sazo, Dirk Brömmel, Robert Speck | Jülich Supercomputing Center, Forschungszentrum Jülich


Kepner, Evan (2020). Mutatest 3.1.0 Documentation. Mutatest Documentation. URL: https://mutatest.readthedocs.io/en/latest/install.html#mutation-trial-process (visited on 05/15/2024).


Wacker, Mike (Apr. 22, 2015). Just Say No to More End-to-End Tests. Google Testing Blog. URL: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html (visited on 10/11/2023).