



Scientific  
Software  
Center



UNIVERSITÄT  
HEIDELBERG  
ZUKUNFT  
SEIT 1386

# Effective Software Testing

---

Liam Keegan, SSC

# Course Outline

- Testing
  - Benefits of a good test suite, difficulties of creating one
- Types of tests
  - Different types of tests, their use cases
- Best practices
  - How to write good tests
- Testing strategies
  - Strategies for testing new and legacy code
- Hands on with pytest
  - Test driven development with pytest

# Testing

---

# What is testing anyway?

There are many ways in which all software is tested:

- It is “tested” every time it is used and produces some output
- It was probably tested with some sample inputs when written
- There is probably some manual testing when changes are made

However this kind of “testing” can quickly become both insufficient and inefficient as a software project grows in complexity.

Changing the code risks breaking things that previously worked without notice, and manually testing all the functionality quickly becomes an impossible task

# Automated tests

Many software projects also have automated tests

- A test is a piece of code that tests some behaviour of the software
- A test suite is a collection of such tests
- The test suite typically runs automatically whenever a change is made
- The more complex the project, the more value a good test suite provides
- But a test suite is not only for large projects!

This is the kind of testing we will learn about in this course.

# A good test suite provides many benefits

- Ensure **correctness** of your code when you write it
- **Maintain** correctness of your code as things change around it
- Make changes or **refactor** code without fear
- Find bugs **earlier** and more easily
- Easier for new contributors to make **positive** changes
- Complements the **documentation** as examples of use
- Gives others **confidence** in the correctness of your code
- Encourages **well-designed** modular code and interfaces

# So why doesn't every project have one?

- Requires upfront investment to create
- Changing code (often) also requires changing tests
  - Mitigated by writing good tests that are not brittle
- Slows (initial) speed of development
  - But speeds up later development and improves quality
- Hard to retrofit to legacy code
  - Approval testing strategy can help
- Bad tests can be worse than no tests
  - False negative tests can waste time or result in test failures being ignored
  - False positive tests can give false sense of security

# Motivating example

- Imagine you have an existing, working software project
  - You want to add a small feature
  - You inadvertently break some other functionality with your changes
- With only manual tests
  - You don't find this bug with your manual testing of the new functionality: Bad!
- With a brittle test suite
  - The tests for the broken functionality fail: Good!
  - So do a bunch of unrelated tests for unrelated reasons: Bad!
  - Either you waste a lot of time fixing all these unrelated test failures & find the bug
  - Or you ignore them and miss the actual bug in all the noise
- With an effective test suite
  - The tests for the broken functionality fail & you find the bug: Good!



# Types of Tests

---

# Types of tests

- Unit tests
- Integration tests
- System tests
- Regression tests
- Approval tests
- Acceptance tests
- Smoke tests
- Performance tests
- Fuzzing tests
- ...

# Unit tests

- Small, self-contained test of a piece of functionality
- Narrow scope: typically a class or method
- Fast to run
- Doesn't depend on any other components
  - Dependencies sometimes replaced with mocks or doubles
- Typically most (e.g. 80%) of your tests should be unit tests
- Primary way of testing correctness
- Should be written alongside the functionality being tested
- Failing unit test directly tells you what has gone wrong

# Integration / System tests

- Also known as “end-to-end” or “functional” tests
- Tests involving multiple interacting components
- Should be a smaller fraction (e.g. 20%) of your test suite
- Compared to unit tests
  - Typically take longer to run
  - Typically more risks of being brittle or flaky
  - Typically need more maintenance
- If most tests end up as integration tests, consider making code more modular
- Larger projects may have a hierarchy of these
  - Different categories of size / complexity of tests, e.g.
  - Integration tests < Functional tests < GUI tests < End-to-end tests

# Regression tests

- Tests written when a bug is found
- Can be unit or integration tests
- They initially demonstrate the existence of the bug
- Once the bug is fixed, they ensure it doesn't come back
- Ideally: make a PR with two commits
  - First commit adds the regression test, CI now fails
  - Second commit fixes the bug, CI passes
- Note:
  - The phrase “regression testing” is also used to describe the process of running the test suite after changing the code to check for any regressions (failing tests)

# Approval tests

- Useful when dealing with legacy code that lacks tests
- Run a function, record the output, test that it gives this output
- This is not how you would write a test for new code!
- These are not testing for correctness, only for consistency
- But quick to create and can be done without deep understanding of the code
- Then you can start to refactor or make changes
- At least you get a test failure when the outputs change
- Note:
  - “approval testing” is also used to describe the tests a customer might make of a piece of software they commissioned to approve that it fulfils their requirements

# Performance tests

- Test performance using benchmarks for specific workloads
- e.g. CPU time taken to perform a benchmark task
  - Typically need dedicated hardware for this to be reliable
- e.g. perform a task with
  - High CPU load
  - Large amounts of data
  - Many concurrent requests
  - Etc
- Fail criteria can include
  - Incorrect output
  - Benchmark being significantly slower than previously

# Smoke tests

- Basic sanity check: does it run without crashing?
- Electronics analogy
  - is there smoke when you switch it on?
- Plumbing analogy
  - fill pipes with smoke, does it escape?
- Can be a useful pre-test test suite
  - If this fails don't need to run the expensive full test suite



# Fuzzing tests

- Fuzzers are testing tools that generate many random inputs
- They call your code with these inputs and try to cause problems
- Typically used for low-level C++ libraries
- Nowadays also used for Python libraries
- C++: [llvm.org/docs/LibFuzzer.html](http://llvm.org/docs/LibFuzzer.html)
- Python: [pypi.org/project/atheris](http://pypi.org/project/atheris)

# Good tests are...

- **Correct**
  - They test that the thing they are testing is working
- **Readable**
  - It is obvious from looking at it what the test does
- **Complete**
  - They covers all relevant cases and behaviours
- **Documentation**
  - They demonstrate how the code being tested should be used
- **Resilient**
  - They only fail when the thing being tested is false, not for any other reason
- **Unchanging**
  - They don't need to be modified unless the behaviour being tested changes

# Can a test be unchanging?

- Types of changes to the software:
  - Refactoring: internal implementation changes
  - New feature: add some new behaviour
  - Bug fix: fix a bug that was found
  - Change behaviour: change the existing behaviour
- Ideal case: only changes in behaviour should require tests to be updated
  - Refactoring: no change to existing tests, no new tests
  - New feature: no change to existing tests, add new tests for new behaviour
  - Bug fix: no change to existing tests, add new tests for bug
  - Change behaviour: update existing tests

# Best Practices

---

# Testing Pyramid

## Integration tests

Compared to unit tests:

- Slower
- More brittle
- Less precise



## GUI / manual tests

Compared to integration tests:

- Slower
- More brittle
- Less precise

## Unit tests

The core of your test suite

- Fast
- Robust
- Precise

# Don't test unrelated things

- Don't assert things unrelated to the thing you are testing
- Avoid assumptions about the internal structure of the code

## Why?

- Avoid the test becoming brittle / noisy
  - Unrelated changes should not cause the test to fail
- Make the test more maintainable
  - Unrelated changes should not require the test to be updated
- Make the meaning of the test clear
  - Test failure should tell you what broke and what needs to be fixed

# Test using public APIs

- Tests should use the same interfaces as user code
- Tests should not use or rely on private implementation details
- Also known as “black box” vs “white box” testing

## Why?

- Avoids needing to update tests when internal implementation changes
- More realistic / representative use of code
- Tests can serve as examples of use / documentation
- Encourages good API design

# Test state, not interactions

- A test can check state: state of the system after some actions (i.e. what)
- Or it can check interactions: which actions did the system do (i.e. how)
- It is better to test state than interactions

## Why?

- Less brittle to check **what** happened than **how** it happened
- Less likely to depend on internal implementation details
- Avoids needing to update tests when internal implementation changes



# Test behaviours, not methods

- Having a test for a method often involves testing multiple behaviours
- Better to have a separate test for each behaviour
- E.g. *given* state X, *when* action Y, *then* state Z

## Why?

- Keep meaning of a test clear
- Avoid complexity of test for a method growing over time
- See also BDD: “behaviour driven development”

# Keep test code simple

- Test code should be “obvious upon inspection”
- Should be complete: contain enough information to understand the test
- Should be concise: don’t include irrelevant information
- Avoid “clever” code, complex control flow, magic numbers, etc
- Some code repetition between tests is ok if it makes test code simpler

## Why?

- There are no tests for your tests!
- When a test fails, reading the test code should tell you what is wrong

# Aim for completeness

- Generally impossible to test all possible combinations of inputs
- But many inputs are equivalent in terms of the resulting code path
- Attempt to identify “equivalence classes” and test one from each
- Often the edges of a range are worth testing
- For sequences, a good starting point: 0 elements, 1 element, many elements
- Also “interesting” (in your opinion) values or edge cases

## Why?

- We can only prove that code is incorrect with a failed test
- No number of passed tests can prove correctness
- So we need to do our best to create tests that can fail

# Aim for validity

- Ensure that every test can fail!
- Avoid circular logic
  - e.g. same code in test as in implementation
  - This test is assuming the implementation is correct and will always pass
- Use appropriate numerical conditions
  - E.g.  $3 \leq \text{pi}() \leq 4$  will pass for many outputs that may be unacceptable
  - But  $\text{pi}() == 3.1415926535897$  may fail for an output that was actually ok
  - Often a sensible choice here depends on your use case
- Often worth testing the test
  - Intentionally (temporarily) break the implementation in some way
  - Check the test actually fails!

# Name tests well

- Test names should include the behaviour being tested
- Seeing the failing test name should already give a good idea what is broken
- It is fine if this makes the test name long
  - We're not **calling** this function in our code, it being long doesn't matter
  - We're **reading** its name in a failing test report, a human should understand its intent
- Some examples: bad short name -> better longer name
  - `test0` -> `test_divide_by_zero_raises_exception`
  - `test_auth` -> `test_invalid_user_should_deny_access`
  - `test_widget` -> `test_mouse_click_on_widget_changes_colour`
- Consider a sentence involving “should” as a starting point for the name
- Try to ensure consistency in test naming

# Testing Strategies

---

# Scenario: New code

- Write unit tests as you write code (test driven development)
  - TDD: first write a failing test, then write code so that it passes, then refactor
  - Alternative: first write a small piece of code, then a test for that code
  - Less good: write a bunch of code, then later add a bunch of tests
- Benefits
  - Makes you think about the API before / as well as the implementation
  - Forces you to actually use the API of the code you just wrote
    - If writing unit tests using this API is difficult, maybe the design could be better?
  - Tests are not added as an afterthought
  - Writing a failing test before the code to make it pass ensures the test can actually fail

## Scenario: Legacy code

- You have inherited a large codebase without tests
- How do you safely extend or modify it without inadvertently breaking things?
  - Adding unit tests for everything is typically unrealistic
  - You might not even know what much of the code does
- One strategy here is to generate approval tests
  - Run the code with some valid input, save the output
  - Write a test that does this and checks that the output is still the same
  - Like this you can quickly construct a test suite
  - These are not very good tests, but they are much better than nothing!
  - You then at least have a warning when a change you make modifies the existing behaviour
- Your new code should have unit tests as normal



## Scenario: Jupyter notebooks

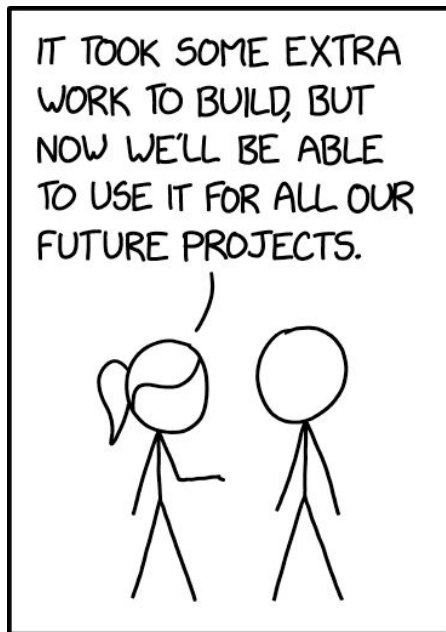
- What about if you write code in jupyter notebooks?
- You can use the [ipytest](#) package to run tests inside the notebook
  - `pip install ipytest`
- Write your pytest test case inside a notebook cell
- Add this command to the first line of the cell:
  - `%%ipytest`
- Executing the cell will run pytest & display the output below the cell

A big advantage of this over just manually running code in a cell to test things is that if you later transfer this code into a python module or package you can also transfer the tests.

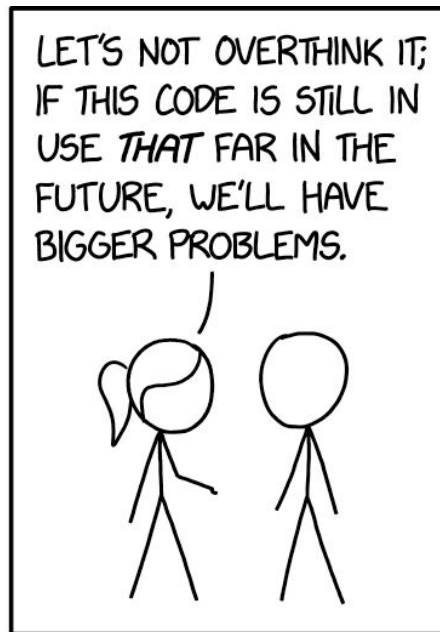
# Scale and future-proofing

- In early stages of development when the project is small
  - Writing unit tests typically slows down development for limited benefit
  - Manual tests are often sufficient
- As the project progresses
  - Eventually it becomes apparent that tests are required
  - But retrofitting unit tests would now be a huge job...
  - Could just add some integration/approval tests...
  - Notice how your new project is already looking like legacy code!
- General strategy
  - Bear in mind the possible future use and development of your code
    - If you are certain there is none, probably ok to skip tests
    - In most other cases, writing unit tests alongside code from the start is a good default strategy
- Watch out for this pattern:
  - Write some “throwaway code” without tests, it works, you use it...
  - ...code turns out to be useful for something else, is modified, extended...
  - ...becomes unmaintainable, oops, this is legacy code!

# Scale and future-proofing



HOW TO ENSURE YOUR CODE IS NEVER REUSED



HOW TO ENSURE YOUR CODE LIVES FOREVER

# pytest

---

# Pytest

- pytest is a widely used Python test framework
- Makes it easy to write small and readable tests
- Also offers more advanced features such as fixtures and mocks
- Large ecosystem of plugins providing additional functionality
- Well documented: [docs.pytest.org](https://docs.pytest.org)
- Simple installation
  - `pip install pytest`
  - `conda install pytest`
- Quick overview:
  - [ssciwr.github.io/lunch-time-python/lunchtime4/lunchtime4.slides.html](https://ssciwr.github.io/lunch-time-python/lunchtime4/lunchtime4.slides.html)

# Pytest in one slide

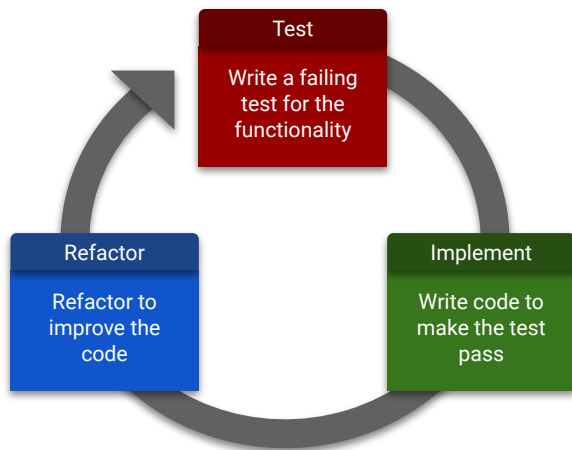
- Install it
  - `pip install pytest`
- For every file `x.py`, add a file `test_x.py` in a folder called `tests`
  - `test_player.py`
- In this file, write functions that start with `test_`
  - `def test_player_initial_age_zero():`
- Assert things inside these functions about your code
  - `player = Player()`
  - `assert player.age == 0`
- Run pytest
  - `python -m pytest`
- You now have an automated test suite!

# Hands on TDD with pytest

---

# Hands on with pytest

- Start working on a simple tic-tac-toe game in Python
- We'll develop some (very) basic functionality together
- Do this in a TDD (test-driven-development) style





# Starting point

- Start from a mostly empty project
- Clone the repository from github:
  - `git clone https://github.com/ssciwr/effective-software-testing.git`
  - `cd effective-software-testing`
- Checkout the “cleanstart” branch:
  - `git checkout cleanstart`
- Do an editable install of the package:
  - `python -m pip install --editable .[tests]`
- Run the tests (note that there aren't any yet!)
  - `python -m pytest`

# Feature roadmap

- Starting point: we have a **P**layer
- Implement a **B**oard to store the game state
- Allow a player to make a move on a square of the **B**oard
- The **B**oard validates the moves and only allows valid ones
- The **B**oard determines if a game is over and which **P**layer won
- Implement an **E**ngine that can play the game
- Implement a GUI interface to play against the **E**ngine

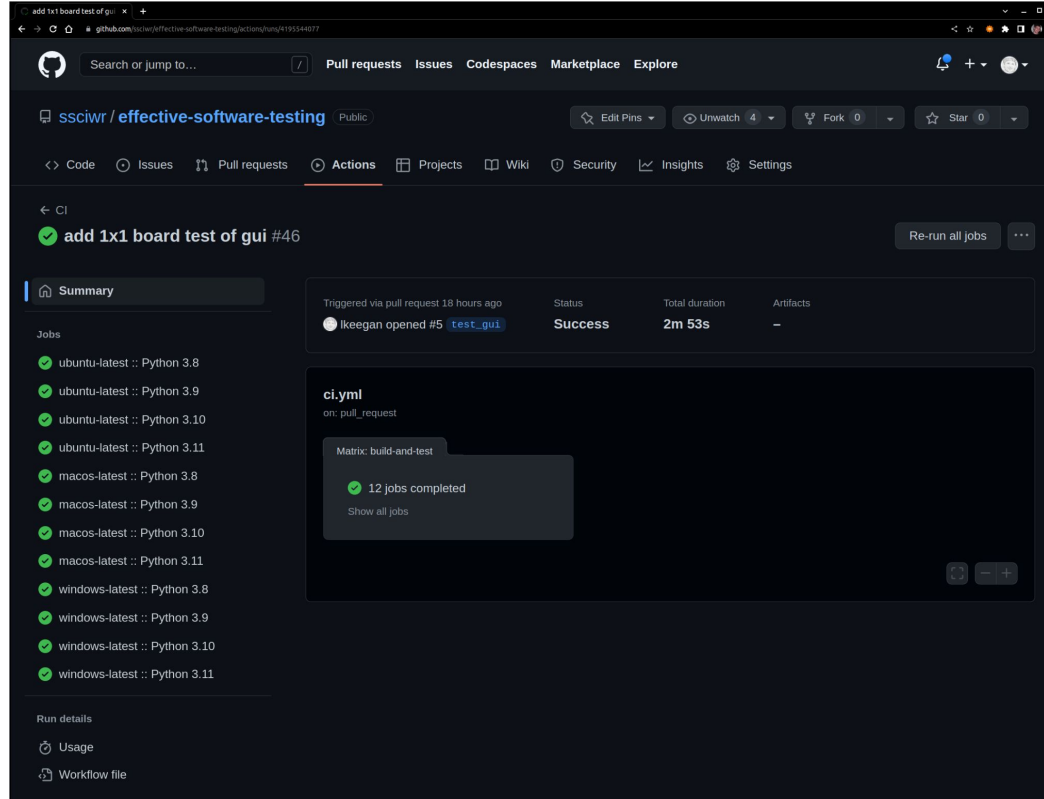
(Actual goal: the real goal here is of course not to implement these features, but instead to apply the concepts and best practices we have been discussing to create the test suite for this project!)

# Coding time!

# Complete project

- Let's skip ahead to the completed implementation
- Change to the main branch:
  - **git checkout main**
- (re)-install the package:
  - **python -m pip install --editable .[tests]**
- Should now see more tests when we run pytest:
  - **python -m pytest**
- Hopefully you can also play the game:
  - **tic-tac-toe**

# PR: Continuous Integration



The screenshot shows the GitHub Actions interface for a pull request. The workflow is named "add 1x1 board test of gui #46" and is in a "Success" state. It was triggered by a pull request 18 hours ago. The total duration is 2m 53s. The workflow file is named "ci.yml" and is located in the ".github/workflows/" directory. The workflow matrix includes 12 jobs, all of which are completed. The jobs are organized by operating system and Python version.

**Summary**

Triggered via pull request 18 hours ago

Status: **Success**

Total duration: 2m 53s

Artifacts: -

**ci.yml**

on: pull\_request

Matrix: build-and-test

12 jobs completed

Show all jobs

**Jobs**

- ubuntu-latest :: Python 3.8
- ubuntu-latest :: Python 3.9
- ubuntu-latest :: Python 3.10
- ubuntu-latest :: Python 3.11
- macos-latest :: Python 3.8
- macos-latest :: Python 3.9
- macos-latest :: Python 3.10
- macos-latest :: Python 3.11
- windows-latest :: Python 3.8
- windows-latest :: Python 3.9
- windows-latest :: Python 3.10
- windows-latest :: Python 3.11

**Run details**

- Usage
- Workflow file

# PR: Code Coverage

add 1x1 board test of gui #5

Merged

lkeegan merged 1 commit into `main` from `test_gui` 18 hours ago

No description provided.

add 1x1 board test of gui

Verified ✓ a613679

codecov bot commented 18 hours ago • edited

## Codecov Report

Merging #5 (a613679) into main (f7bb116) will increase coverage by 9.86%.  
The diff coverage is n/a.

	Coverage	Diff
Files	7	7
Lines	172	172
+ Hits	144	161 +17
+ Misses	28	11 -17

Impacted Files	Coverage Δ
effective_software_testing/board_widget.py	96.66% <0.00%> (+1.66%)
effective_software_testing/gui.py	86.28% <0.00%> (+55.17%)

Help us with your feedback. Take ten seconds to tell us [how you rate us](#). Have a feature suggestion? [Share it here](#).

lkeegan merged commit 592a693 into `main` 18 hours ago  
14 checks passed

View details Revert

lkeegan deleted the `test_gui` branch 18 hours ago

Restore branch

No reviews

Assignees  
No one—assign yourself

Labels  
None yet

Projects  
None yet

Milestone  
No milestone

Development  
Successfully merging this pull request may close these issues.

None yet

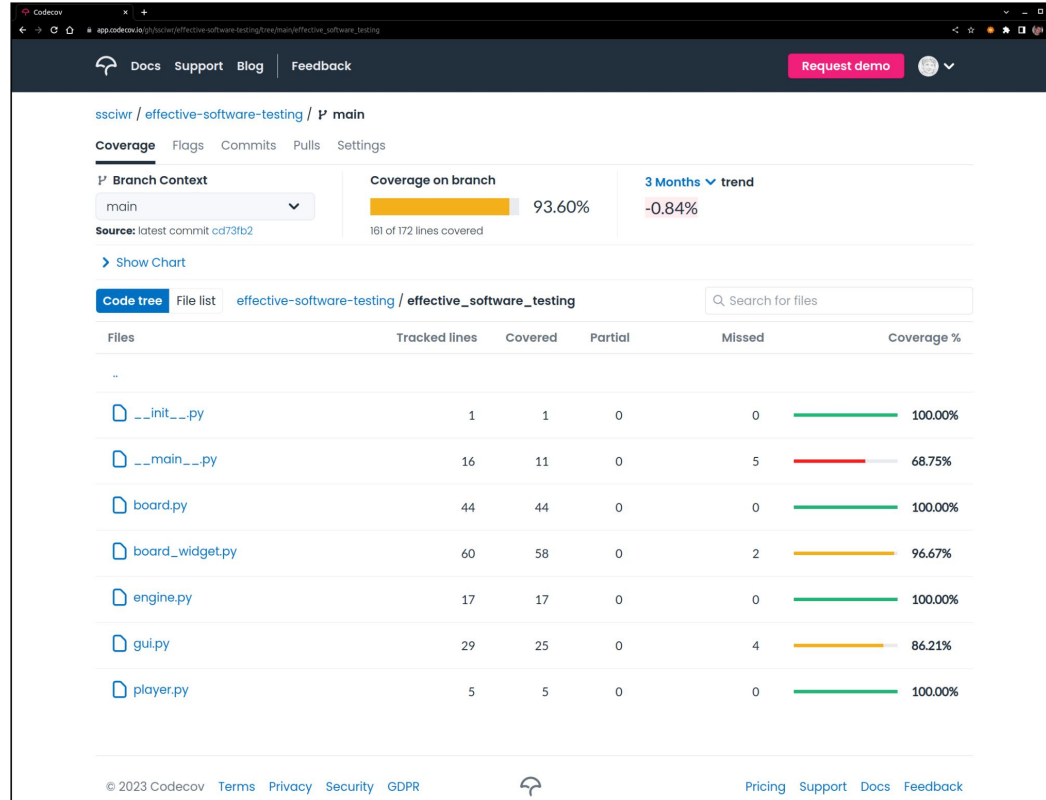
Notifications  
Customize  
Unsubscribe

You're receiving notifications because you're watching this repository.

1 participant

Lock conversation

# Code Coverage



# Add a feature

- Desired feature
  - Board should also tell us the start/end location of winning n-in-a-row
    - E.g.  $(0, 0)$ ,  $(0, n-1)$  for a full first row
  - GUI should then display a line showing this when someone wins
- Workflow
  - Create a new feature branch
  - Implement this feature and the corresponding tests
  - Make a pull request to the main branch with these changes
  - Check that CI and code coverage passes
  - Merge the changes



# Coding time!

# Summary

---

# Summary

In this course we covered:

- The benefits of having a test suite
- The different types of test
- Best practices for writing good tests
- Testing strategies for new and legacy code
- Hands on TDD (Test Driven Development) with pytest
- Hands on Github / Continuous Integration workflow

# Next steps

- “Software Engineering at Google” book
  - Free html version online
  - Three chapters on testing
  - <https://abseil.io/resources/swe-book/html/toc.html>
- Your next Python project
  - [github.com/ssciwr/cookiecutter-python-package](https://github.com/ssciwr/cookiecutter-python-package)
    - Generate a Python project with pytest tests, CI, coverage, etc
- Your next C++ project
  - <https://github.com/ssciwr/cookiecutter-cpp-project>
    - Generate a C++ project with catch2 tests, CI, coverage, etc
    - [github.com/catchorg/Catch2](https://github.com/catchorg/Catch2) is a simple and widely used c++ testing framework