It work’s on my machine

A workshop on software testing

Thomas Madlener

Apr 26, 2024

Menu for today

- Basics of testing (theory) and a few buzzwords
- Introduction to `pytest` and `Catch2` frameworks
- Some bits about organizing code / packaging
- Some pointers for setting up CI
- Exercises (and food for thought)

What not to expect

- Introduction to c++ / python, git or cmake
    - See the HSF Training Courses for that
- An indepth lecture about all things to know about testing
- A complete guide on pytest or Catch2
    - Check the docs pytest, Catch2
- A complete guide on setting up CI
- A complete guide on packaging
- An overview of available unittest frameworks

Why write tests at all?

- “This worked on my machine!”
- “Ah yeah, this issue, we fixed it a few months ago. I wonder why it’s not working now?”
- “I only changed this one line, why is everything broken now?”
- “Why is debugging so hard? Isn’t there an easier way?”

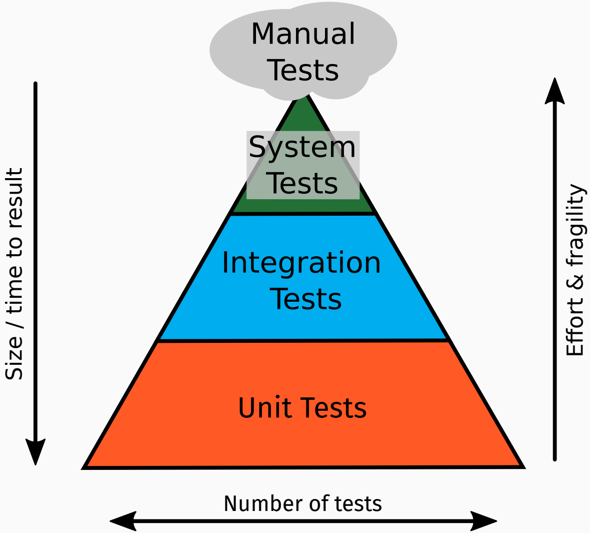
What tests can do for you

- Confidence that things work (and keep working)
  - Enabling refactoring without breaking things
- *Shift left* in the debugging process
  - Everything that has a **meaningful test** can be excluded from debugging
- More modular code
- Builtin (and up to date) examples and documentation for your code
- Nice status badges on your repository (if you run them in CI)

What it (unfortunately) can't do

- Make sure there are no bugs in your code at all

The testing pyramid



- Focus on unit tests today
- Only unit tests is not enough for a working system
- Only integration / system tests are hard to maintain and interpret
- **Unit tests are**
  - quick to run
  - small in scope
  - independent of each other
  - large in numbers

My first unittest

Code to test

```python
def add_one(number):
    return number + 1
```

```cpp
int add_one(int number) {
  return number + 1;
}
```

Unit test

```python
def test_add_one():
    assert basics.add_one(3) == 4
```

```cpp
TEST_CASE("add_one") {
  REQUIRE(basics::add_one(3) == 4);
}
```

python (pytest)

C++ (Catch2)

Anatomy of a unit test

- (Arrange)
  - Setup things that you want to test (if necessary)
- Act
  - Run the function that you want to test
- Assert
  - Check that the results of the function are as expected
- (Cleanup)
  - Cleanup resources (if necessary)

Writing testable code

```python
def monster_function(fn, options):
    # ----- collect data -----
    # ... (20 lines of code)
    # ... I mean it just takes a bit to do that
    # ...



    # ----- calculate stuff -----
    # ... (50 lines of code)
    # ... Obviously these are non-trivial calculations
    # ... They require many comments to explain as well
    # ... So many lines of code
    # ...



    # ----- make plots -----
    # ... (50 lines of code)
    # ... We could do this in 10 lines maybe?
    # ... But we want nice plots
    # ... We just have to write this once
    # ... Then we simply copy it to the next function
    # ... I wonder if there is a better way to do this
    # ...

    # ----- write output file -----
    # ... (20 lines of code)
    # ... Otherwise what's the point?
    # ...
```

Writing testable code

- Small functions / classes
- Functions / classes with a single purpose
- “How can I test this?”

```python
def collect_data(fn):
    # collects data
    return data

def calc_stuff(data, options):
    # calculate result
    return result

def make_plots(data, result):
    # make plots
    return plots

def write_output(plots, fn):
    # write the output

def monster_function(fn, options):
    data = collect_data(fn)
    res = calc_stuff(data, options)
    plots = make_plots(data, res)
    write_output(plots, fn + ".out")
```

Writing testable code

- Small functions / classes
- Functions / classes with a single purpose
- “How can I test this?”
- Side benefits:
    - Reusable code
    - Easier to maintain
    - Easier to understand

```python
def collect_data(fn):
    # collects data
    return data

def calc_stuff(data, options):
    # calculate result
    return result

def make_plots(data, result):
    # make plots
    return plots

def write_output(plots, fn):
    # write the output

def monster_function(fn, options):
    data = collect_data(fn)
    res = calc_stuff(data, options)
    plots = make_plots(data, res)
    write_output(plots, fn + ".out")
```

Practical considerations for unit tests

- Also test the *error path*, not just the *happy path*

```python
def foo(a):
    if not isinstance(a, int):
        raise ValueError()
    return a * 2
```

```python
def test_foo_invalid_input():
    with pytest.raises(ValueError):
        foo(3.14)
```

python (pytest)

```cpp
int bar(int a) {
  if (a == 0) {
    throw std::runtime_error("");
  }
  return 42 / a;
}
```

```cpp
TEST_CASE("bar invalid input") {
  REQUIRE_THROWS_AS(bar(0),
    std::runtime_error);
}
```

C++ (Catch2)

Practical considerations for unit tests

- Floating point comparisons are *hard* (see e.g. python docs)

```python
from pytest import approx

def test_floats():
    # This will fail!
    assert 0.1 + 0.2 == 0.3
    # This will pass
    assert 0.1 + 0.2 == approx(0.3)
```

python (pytest)

```cpp
using namespace Catch::Matchers;

TEST_CASE("float comparison") {
  // This will fail
  REQUIRE(0.1 + 0.2 == 0.3);
  // This will pass
  REQUIRE_THAT(0.1 + 0.2,
               WithinAbs(0.3, .1));
}
```

C++ (Catch2)

- Need to choose required / desired accuracy up-front!

Practical considerations for unit tests

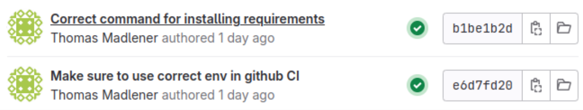
- *Test fixtures* allow you to create inputs for tests
  - Handy if the setup requires a few steps
  - Useful when tests require resources (db connection, some running server, …)
- Unit tests should be **very** easy to read and understand
- Using random inputs for unit tests is possible but the assertions are hard to get right
  - Even statistial tests fail from time to time (by design)
  - It can be an extremely good way of finding bugs (*Fuzzing*)
- A *flaky* test quickly becomes useless
- Ignored tests are (almost) worse than no tests

General considerations for tests

- Try to group tests
    - Allows you to run only “interesting” tests during development
- Turn issues into test cases
    - Invest the time to create a minimal reproducer
    - *Red-Green Testing*
- Almost anything that can be automated / scripted can be turned into a (integration / system) test
    - (Usually) trivial to integrate into CI
- Some things don’t need tests

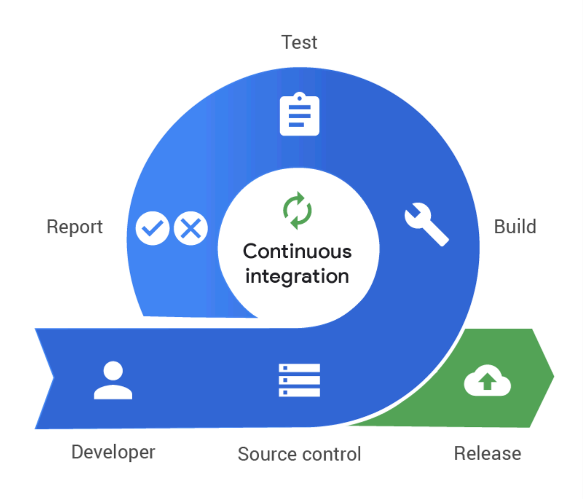
Continuous Integration - CI

- Frequently integrate code changes into main branch
- Quick feedback loop
- Often synonymous with “having tests that run automatically on repository”



Correct command for installing requirements
Thomas Madlener authored 1 day ago
b1be1b2d

Make sure to use correct env in github CI
Thomas Madlener authored 1 day ago
e6d7fd20



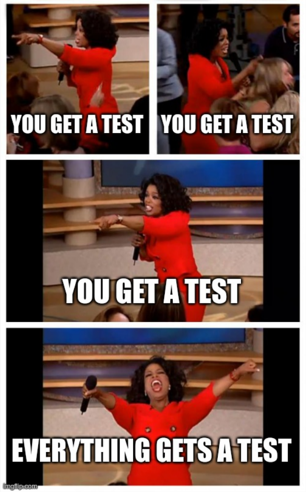
CI - Some technicalities

- Available on gitlab (GitLab CI/CD) and github (GitHub Actions)
- Very similar working principles
  - YAML as config language (but different grammars)
- See our introduction for the basics of GitLab CI/CD
- Exercises have configuration for both
  - Push to github if you want to see github actions
  - Slightly different approaches for dependencies

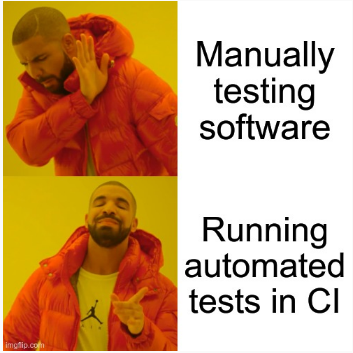
Considerations for CI

- You can run more stuff than just tests in CI
    - E.g. static analyzers, linters, documentation generation & deployment
- Keep the feedback loop as short as possible
    - Consider stages for running quick tests first
- Run on different platforms / compilers, ...
- Keep dependencies stable
- Include enough output to diagnose the problem quickly
    - Consider storing artifacts that can be inspected
- Make it possible to reproduce the CI environment locally

Summary



- (Non-trivial) software requires tests
- Unit tests form the basis
- Test automation enables CI
- Even tested software will have bugs



Exercises

- There are two sets of prepared exercises for either python or c++
- Very similar examples to start with, some obvious differences in setup
- The intermediate / advanced examples differ between the two
- Solutions included, you can also do them on your own after the workshop
- Basic configuration for CI already in place
- **Pick one to start with for now**
- **Initial setup interactively together in first half / now**
- Second half of workshop to work on exercises on your own
    - Call us for help whenever necessary

.desy.de/fh-sustainability-forum/sustainable-coding-tutorial/python-unittesting
.desy.de/fh-sustainability-forum/sustainable-coding-tutorial/cpp-unittesting