



# Robust Programming

Julien Peloton & Hadrien Grasland

2026-04-10



# Disclaimer

- This is an **introductory** course
  - You can't become an expert in 1 day
  - But you can learn the general process + simple know-how
  - Open to adding advanced courses: suggest topics!



# Before starting

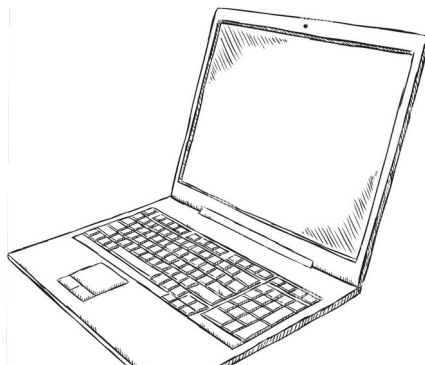
- Quick and dirty development, we all know how to do it...
- The first moments are often intense, and very rewarding.
- ... but the happiness rarely lasts forever:
  - Large changes become harder to make over time.
  - At some point the code is so messy that we consider **(1) rewriting it, (2) restarting from scratch, (3) giving up.**
- What went wrong?

# What went wrong?

- **Not the quality of the people**
  - Average talent level is about the same :-)
- Often problems we **did not anticipate**
- Also boring **technical debt**:
  - poor documentation;
  - put off unit and integration testing;
  - LLM-generated code you don't understand.
- This doesn't explain all of it; but a large part of it.

# Our strategy

1. Automatic rule checkers
2. Continuous integration
3. Documentation
4. Tests





# Our strategy

- 1. Automatic rule checkers**
2. Continuous integration
3. Documentation
4. Tests

# If you were talking...

- Many ways to convey a message...
  - Yo – wassup / *Hello, how are you?*
  - LGTM / *The newly pieces of code follow our conventions, and I agree to merge it to the rest of the codebase*
- ...but some are easier to understand than others ;-)
  - Also true of programming languages!



# Rule checking

- Static analysis = check programs without executing them.
- Typically used to enforce common coding conventions
  - check that coding style is respected
  - perform type checks.
- **Uniformity matters more than any specific style choice.**



# Our strategy

1. Automatic rule checkers
- 2. Continuous integration**
3. Documentation
4. Tests

# On the rise of forges



- A forge is an online tool that typically provides:
  - Hosting capabilities
    - Code, static web site, wiki, Docker images (registry)
    - On a public server, or self-hosted
  - Code history visualization (tree, versions)
  - Discussion tools : Bug tracker, merge/pull requests...
  - Event notifications, statistics, third-party integration
  - **Continuous integration/deployment services**

# Continuous integration

- Documentation, tests, code linting... If you have to **manually** run them after each code change, you'll quickly **give up/forget!**
- Instead, we advise using **Continuous Integration**.
  - **Automatically** run all boring tasks each time you modify the code.
  - Produce a **summary report** to let you focus on the changes.
- Many possible platforms. In this lecture, we'll use **GitLab CI**.

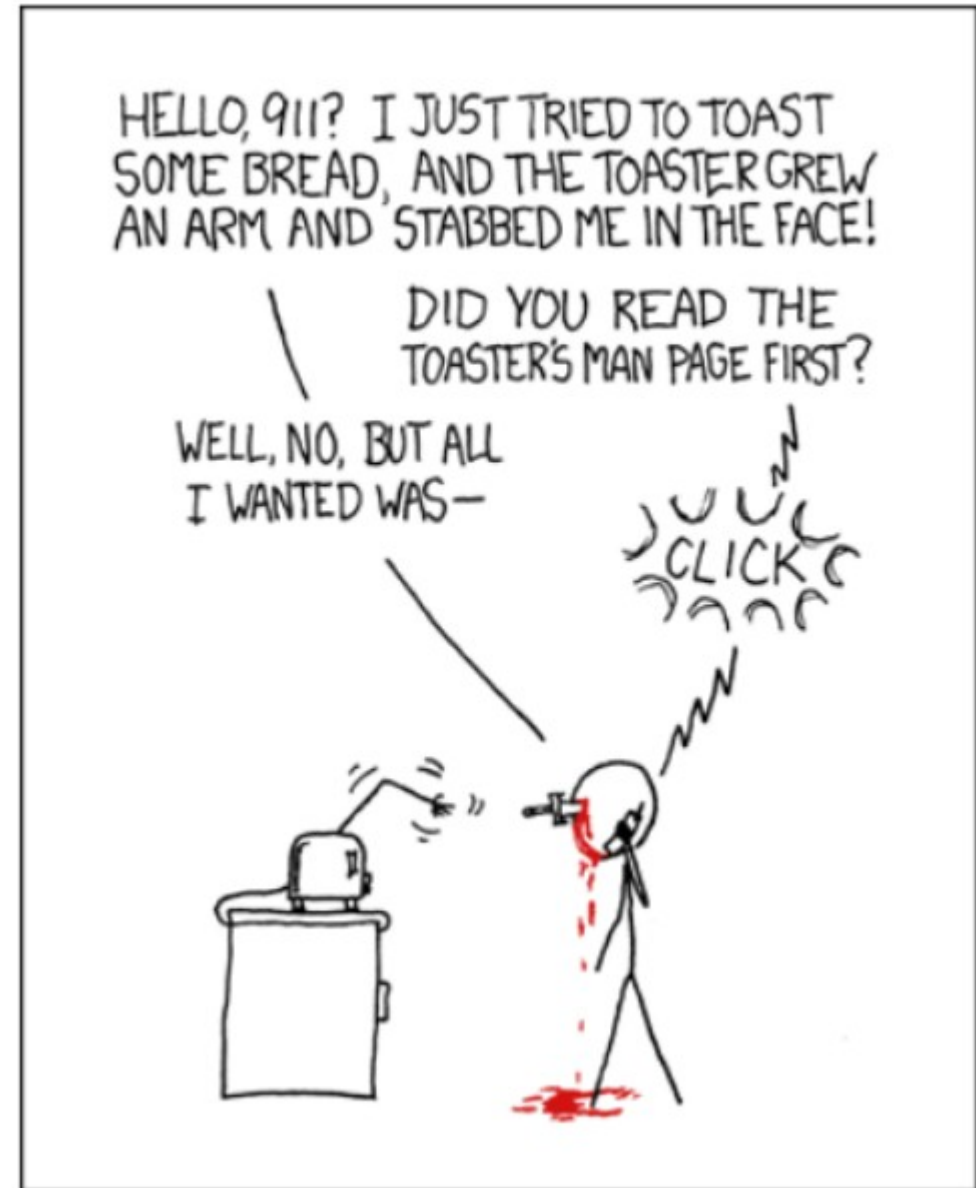


# Our strategy

1. Automatic rule checkers
2. Continuous integration
- 3. Documentation**
4. Tests

# Documentation

- What is it?
  - Specification, code comments, user manual, how-to, tutorials, ...
- When do we start?
  - **Before coding!** Read up on « design by contract » for more



# Well... (true story)

- Documentation is **boring**. Writing help files is even more mind numbing.
- I don't know anyone who reads user manuals except as a **last resort**.
- Most programmers are very **lazy**. Writing comments is just more work.
- Programmers dislike doing things that are not programming. It's an ego thing.
- **Reading the code** is the best way to know how a program works.
- Too many customers require documentation, but **have no clue** on what should go into it. We are programmers, not magicians or mind-readers.
- Documentation and programming are two entirely **different skill sets**
- **Vague requirements** like "...and it has to be documented!". No indication on intended users or usage, nothing on what it should describe.
- Programmers are interested in ideas, and once the ideas are fixed concretely we lose interest in their **communication**.
- Programming is a largely a creative, problem-solving effort. Documenting is largely a teaching and **communication effort**. and so on...



# Why is it rarely done?

- A lot of laziness, but also real barriers:
  - Programming approach that is not only coding
  - Need to know the good practices and be trained
  - Involve communication skills
  - Working with different backgrounds
  - How to value these skills on your scientific career?

# Writing what? And for who?

- **On the code itself**
  - Everything not obvious for someone other than the writer (including the writer one year later).
  - Method pre/post-conditions, planned use for variables, ...
- **Outside the code**
  - User manual, tutorials, online or CLI reference docs, ...
    - Depends on the scope of work: **identify users!**
  - Developers? Internal/external use? Scientists? Everyone?

# Naming is documentation

```
def toto(a, b):  
    """ worst case scenario  
    """  
    return b[a]
```

```
def extract_value_from_dict(key: str, data: dict) -> float:  
    """ better scenario  
    """  
    return data[key]
```



# Summary

- Documentation should be a continuous process, like tests
- Follow widely used style conventions (e.g. PEP8 in Python)
- Create and use templates to ease the writing process
- Generate documentation from the code automatically
- Prefer IDEs or advanced text editors over Notepad
- Automate as much as possible!



# Our strategy

1. Automatic rule checkers
2. Continuous integration
3. Documentation
- 4. Tests**



# Preparing for change

- Any code change is risky
  - May break normal functionality (wrong results!)
  - Today's ideas may turn out to be useless/bad
- How do we prepare for this?
  - **Version control:** Have a way back
  - **Tests:** Find out when you break something



# The perfect test

- **Easy to write:** Little boilerplate, focused on your problem
- **Automated:** Single command, machine-checkable output
- **Realistic:** Close to your real problem
- **Fast:** Can run all basic tests in a couple of seconds
- **Precise:** Narrows source of problem to small code chunks
- **Exhaustive:** Covers most code, over a broad range of inputs
- Some of these goals conflict (e.g. fast/precise vs realistic)



# How to have it all?

- To address contradictory goals, need multiple kinds of tests
  - **Integration/validation tests** close to real world problems
  - **Unit tests** torture individual components (e.g. functions)
- We will specifically focus into **unit tests**
  - Often we use **oracle tests**: for a choice of inputs, compare test output to a known-good value
  - (sort of) **Easy to write**

**Let's try it!**

# Beyond oracle tests

- Problem: Need lots of unit tests to cover all your code
  - If writing tests is tedious/boring, you *will* do it wrong (e.g. code not covered, all tests take same input...)
- Why not offload it to LLMs?
  - Test will pass, but may not be relevant
  - Need independent validation
- A better solution: **property-based testing**

# Property-based testing

- Given a function-like entity to be tested...
  - Generate random inputs
  - Feed them to the function to be tested
  - Check known properties of output
- Much **faster/easier** than manual inputs!
- Generates unexpected inputs → **Exposes assumptions**
- Manual inputs still useful for edge cases, regression testing

**Thanks for your attention!**