Robust Programming

Julien Peloton & Hadrien Grasland

2024-03-28

Disclaimer

- This is an **introductory** course
  - You can’t become an expert in 2 mornings
  - But you can learn the general process + simple know-how
  - Open to adding advanced courses: suggest topics!

Before starting

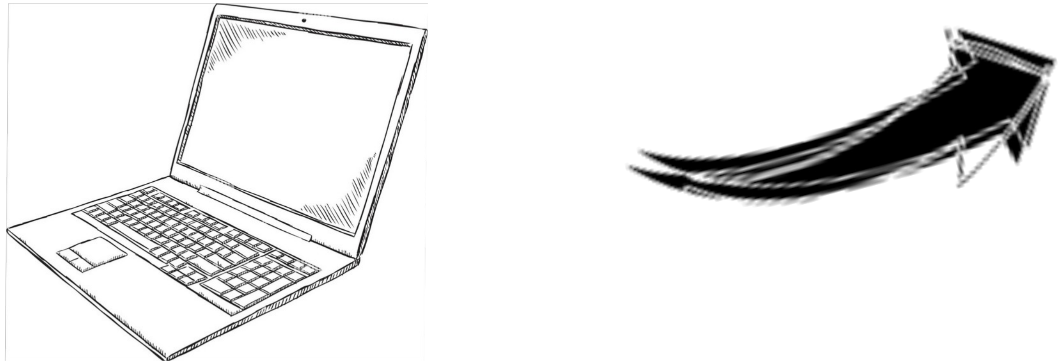
- Quick and dirty development, we all know how to do it…
- The first moments are often intense, and very rewarding.
- … but they rarely make us happy long enough:
  - we often struggle to make similarly large changes over time. Even if the number of contributors grows. At some point the codebase is so messy, that we consider **(1) rewriting it, (2) starting again 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 on problems we **did not anticipate**, which were more important than our ever growing wish list of features.
- Obviously, mostly-boring **technical debt**:
  - poor documentation;
  - deliberately put off unit and integration testing;
  - a lot of manual and redundant actions to perform.
- This doesn't explain all of it; but a large part of it.

Our strategy

1. Automatic rule checkers (Day 1)
2. Documentation (Day 1)
3. Tests (Day 1 & 2)
4. Automation (Day 2)



Our strategy

1. **Automatic rule checkers**
2. Documentation
3. Tests
4. Automation

If you were talking...

- There are many ways to convey a message…
  - Yo – wassup / *Hello, how are you ?*
  - LGTM / *The newly introduced 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 ;-)
  - Why should it be different when it come to programming languages?

Rule checking

- Static program analysis is the analysis of computer programs performed without executing them.
- This is typically used to make sure syntax is uniform across different pieces of code.
  - check that coding style is respected
  - perform type checks.
- Uniformity matters much more than any particular style choice.

Our strategy

1. Automatic rule checkers
2. **Documentation**
3. Tests
4. Automation

Documentation

- What is it?
  - Code specification, code documentation, user manual, how-to, tutorials, …
- When it starts?
  - **Before coding!** E.g. see the programming by contract concept.

HELLO, 911? I JUST TRIED TO TOAST SOME BREAD, AND THE TOASTER GREW AN ARM AND STABBED ME IN THE FACE!
DID YOU READ THE TOASTER'S MAN PAGE FIRST?
WELL, NO, BUT ALL I WANTED WAS—
CLICK

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...

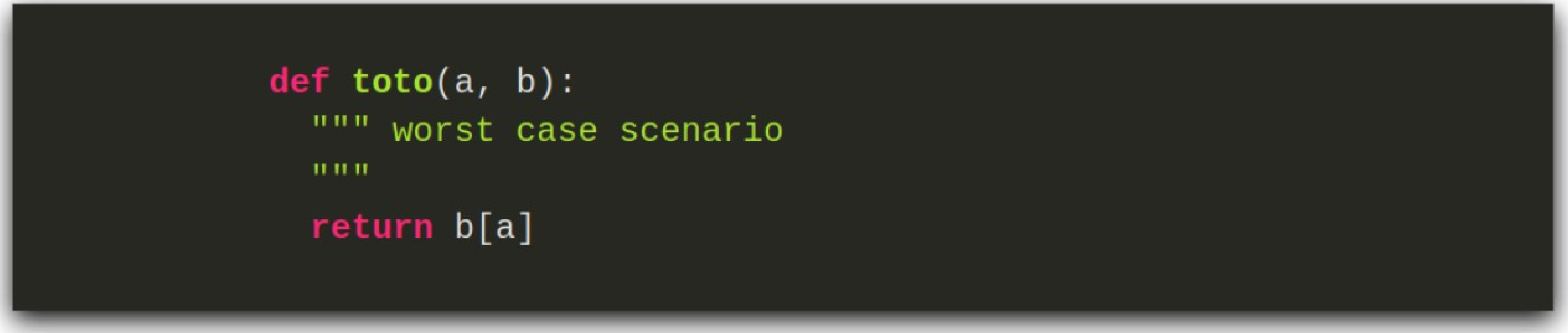
Why is it rarely done ?

- Concretely, 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?

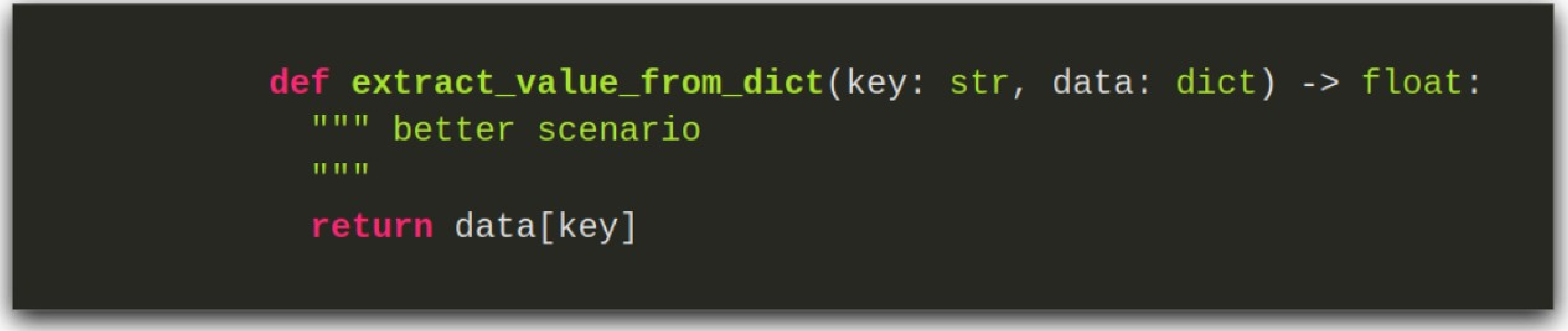
Writing what? And for who?

- On the code source itself
  - Everything that is not obvious for someone else than the writer (including the writer him-herself in a year).
  - In practice pre and post-conditions for the methods, planned use for variables, and everything that can lead to confusion.
- Outside the code source
  - User manual, tutorials, online or CLI documentation, … This will depend on the scope of work: **identify users!**
  - Developers? Internal/external use? Scientific community? General public?

Keep in mind



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



```python
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 style convention from the language (e.g. PEP8 in Python)
- Create and use templates to ease the writing process
- Use tools to generate automatically documentation from the code source
- Use an IDE if you are not a terminal ninja. Refrain using plaintext editor only.
- Automate as much as possible!

Our strategy

1. Automatic rule checkers
2. Documentation
3. **Tests**
4. Automation

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)

Covering the continuum

- To adress 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 first dive deep into the design of **unit tests**
  - Often related to **oracle tests** : for a choice of inputs, we compare the output of the test to a predetermined value
  - (sort of) **Easy to write**

Covering the continuum

- 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…)
  - A good 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

Our strategy

1. Automatic rule checkers
2. Documentation
3. Tests
4. **Automation**

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
  - Visualization of the development (tree, versions)
  - Bug tracker & feature request (including discussion threads)
  - Merge/Pull request (including discussion threads & review)
  - Event notifications, statistics, third-party integration
  - Continuous integration/Continuous deployment services

Continuous integration

- Documentation, tests, code linting... If you had to **manually** run them after each code addition or deletion, you would quickly **give up**!
- Instead, we advice to use the concept of **Continuous Integration**.
  - **Automatically** run all mostly-boring technical tasks each time you modify the code.
  - Produce a **summary report** so that you only have to focus on changes.
- There are many options to set up a continuous integration. In this lecture, we will use the tools integrated with the **GitLab** platform, but there are many other ways!

Thanks for your attention !