IJCLab
Irène Joliot-Curie
Laboratoire de Physique des 2 Infinis

# Robust Programming

Julien Peloton
03/04/2023

# Hello

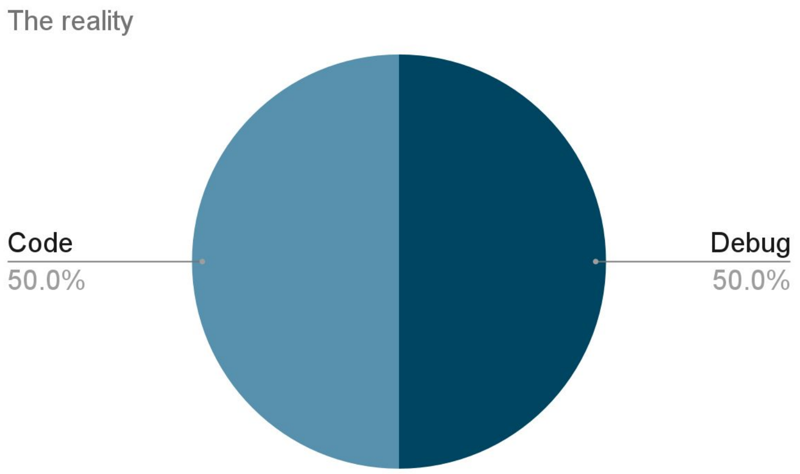
Who am I?

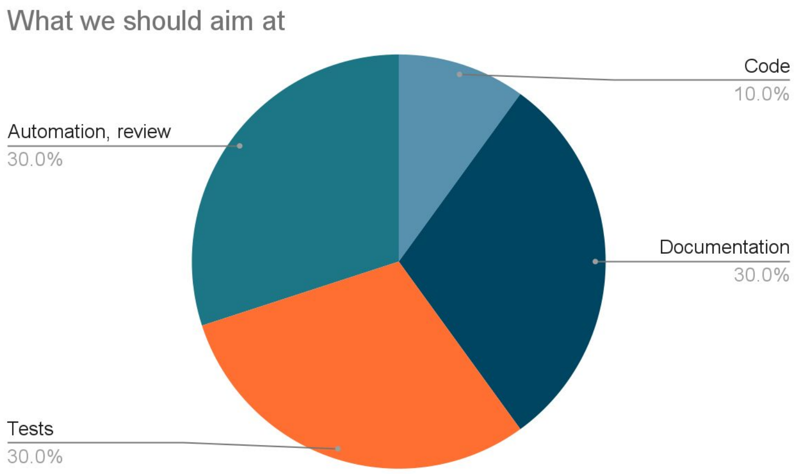
- Joined IJCLab/CNRS (engineer) after PhD/postdoc in cosmology.
- Mainly work in the big data ecosystem nowadays
- Co-lead Rubin broker Fink: responsible for algorithms and infrastructures.
- Helped built various software pipelines in astronomy/cosmology
- Also heavily involved in teaching activities

Views expressed here are my own, sort of.

- Healthy debate at work about what's good and bad about our software practices!
- Everything here has probably been described better in some seminal book I maybe skimmed.

# Let’s face it


The reality
Code
50.0%
Debug
50.0%


What we should aim at
Code
10.0%
Automation, review
30.0%
Documentation
30.0%
Tests
30.0%

# The long run is the difficulty

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.

# 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

# I know, I know...

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 it is 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?

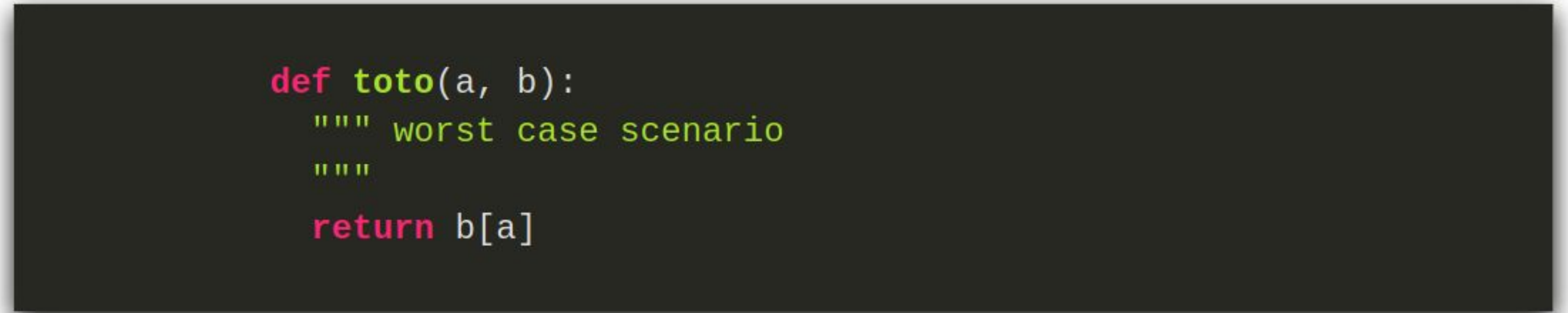
On the code source itself:

- 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.

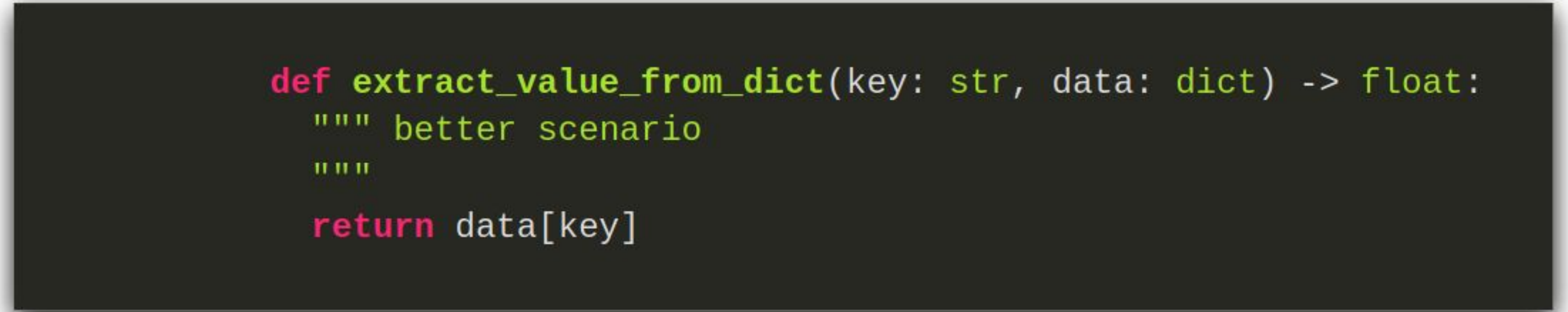
Outside the code source:

- 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?

# Code is also documentation



```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]
```

## hands-on

Go to https://github.com/JulienPeloton/robprog

# Good practices

1. Documentation should be a continuous process, like tests
2. Follow style convention from the language (e.g. PEP8 in Python)
3. Create and use templates to ease the writing process
4. Use tools to generate automatically documentation from the code source
5. Use an IDE if you are not a terminal ninja. Please stop using plaintext editor.
6. Automatise as much as possible!

## Static analysis

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.

In Python, this will be used to:

1. check that Python coding style is respected (PEP8)
2. to perform type checks.

**Uniformity matters much more than any particular style choice.**

# hands-on

Go to https://github.com/JulienPeloton/robprog

# Tests

Testing your code... a very vast subject.

Why we want to test our code?

- Primarily to check the code runs as expected.
- But also to understand the limits and the flaws of our implementation.
- But also to better design our code

When to test? Always!

What to test? As much as possible!

# How to test

Mainly two families of tests: unit tests & integration tests.

There are many tools to help you write a test (doctest, pytest, nose, hypothesis), but eventually you can come up with your own ideas.

The vast majority of tests we write are **oracle tests** – for a choice of inputs, we compare the output of the test to a value predetermined (by us).

- (sort of) Easy to write
- But often limited by our imagination (and laziness)

# Property-based testing

The concept of property-based testing was is borrowed from functional programming.

To summarize, the idea is no longer to predict the output knowing a particular input, but to determine and test the properties of our function. Some properties:

- The invariants
- Inverses (invertible function)
- Idempotence
- Properties of internal composition laws: associativity, commutativity...
- Structural induction (Solve a smaller problem first)
- Hard to prove, easy to verify

# Test-driven development

First write tests, then implement functions.

You should think of property-based testing not as a testing process, but as a design process – a technique that helps you clarify what your system is really trying to do.

- Take the design as given and see what properties you can come up with.

if a key aspect of your design is satisfied with just a simple implementation, then perhaps there is something you have overlooked – something that, when you discover it, will make your design both clearer and more robust.

# hands-on

Go to https://github.com/JulienPeloton/robprog

# Takeaway

- **Test coverage**: Make sure you have a good understanding of the code and ensure that all relevant parts of the code are being tested.
- **Edge cases**: Consider edge cases and test them to ensure that the code is robust and can handle unexpected inputs.
- **Readability**: Make sure that your tests are easy to read and understand, so that other developers can quickly understand what the code is doing.
- **Maintainability**: Keep your tests organized and maintainable, so that they can be easily updated and modified as the code evolves.

And tests should be before all a design process!

## Automatisation

Documentation, tests, code linting... If you had to manually run them after each code addition or deletion, you would give up quickly!

Instead, we advice to use the concept of **Continuous Integration**.

- Tools that will run all mostly-boring technical tasks for you each time you modify the code.
- A summary report will be given 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 GitHub platform, but there are many other ways!

# hands-on

Go to https://github.com/JulienPeloton/robprog

# Release

Versioning is important, not only for you, but for users to figure out the status of the code they are using.

There are many ways to put a version number: Semantic versioning (SemVer), Date of release, custom.

For example in the case of SemVer:

- X[API change].Y[new features].Z[bug fix]
- So 2.2.2, 2.2.3, and 2.3.2 are all compatible among each other, but 3.0.0 is not compatible with others.

## hands-on

Go to https://github.com/JulienPeloton/robprog

# To finish

Code review

- You'll learn a lot reading code from others. Do it!

Open source

- You are all using bits and pieces from all over the world. Do not hesitate to contribute, you will learn a LOT.