From doing math to writing code

Workflow and implementation tips

Mathieu Tanneau

GERAD

March 27, 2019

Thanks to


CIRRELT


GERAD
GROUP FOR RESEARCH
IN DECISION ANALYSIS


RMontreal
student chapter

This talk is about:

- Going from

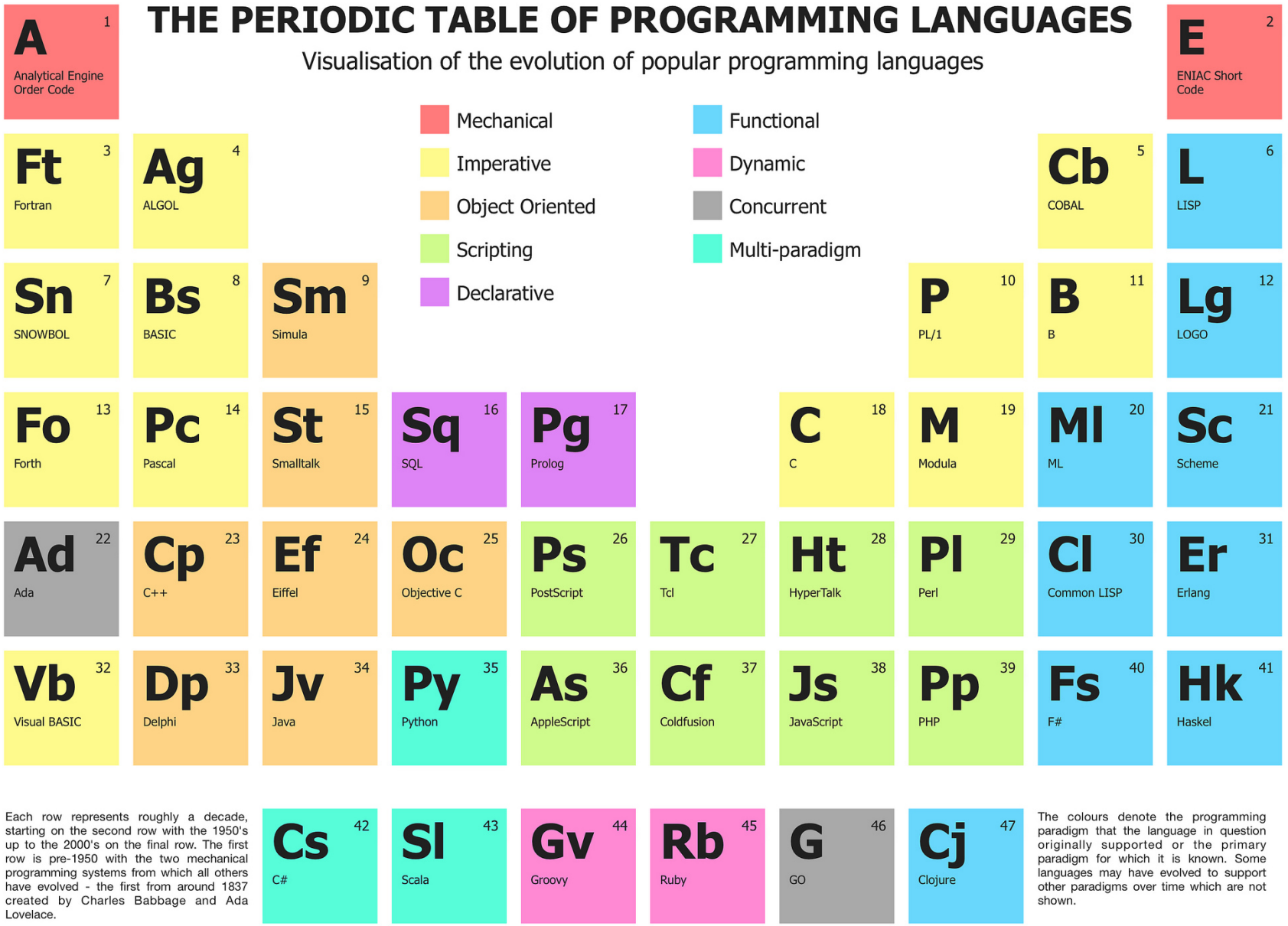
  "the pseudo-code of the method is given in Algorithm 1"

  to

  "computational results are reported in Table 2"
- Making the coding part easier
- Encouraging you to share your code

Material available here: `https://github.com/mtanneau/tutorial_airo`

Outline


THE PERIODIC TABLE OF PROGRAMMING LANGUAGES
Visualisation of the evolution of popular programming languages
Mechanical
Imperative
Object Oriented
Scripting
Declarative
Functional
Dynamic
Concurrent
Multi-paradigm
A 1 Analytical Engine Order Code
E 2 ENIAC Short Code
Ft 3 Fortran
Ag 4 ALGOL
Cb 5 COBAL
L 6 LISP
Sn 7 SNOWBOL
Bs 8 BASIC
Sm 9 Simula
P 10 PL/1
B 11 B
Lg 12 LOGO
Fo 13 Forth
Pc 14 Pascal
St 15 Smalltalk
Sq 16 SQL
Pg 17 Prolog
C 18 C
M 19 Modula
Ml 20 ML
Sc 21 Scheme
Ad 22 Ada
Cp 23 C++
Ef 24 Eiffel
Oc 25 Objective C
Ps 26 PostScript
Tc 27 Tcl
Ht 28 HyperTalk
Pl 29 Perl
Cl 30 Common LISP
Er 31 Erlang
Vb 32 Visual BASIC
Dp 33 Delphi
Jv 34 Java
Py 35 Python
As 36 AppleScript
Cf 37 Coldfusion
Js 38 JavaScript
Pp 39 PHP
Fs 40 F#
Hk 41 Haskel
Each row represents roughly a decade, starting on the second row with the 1950's up to the 2000's on the final row. The first row is pre-1950 with the two mechanical programming systems from which all others have evolved - the first from around 1837 created by Charles Babbage and Ada Lovelace.
Cs 42 C#
Sl 43 Scala
Gv 44 Groovy
Rb 45 Ruby
G 46 GO
Cj 47 Clojure
The colours denote the programming paradigm that the language in question originally supported or the primary paradigm for which it is known. Some languages may have evolved to support other paradigms over time which are not shown.
© 2012 Paul Bowler

Which programming language?

Some will say it's all about performance vs simplicity...

I find these to be more relevant:

- Do you have any hard constraints? (e.g. existing C++ code)
- Which language are you most comfortable with?
- Would it impair you for the rest of your PhD?
- Would it restrict the toolset available to you?
- How big is the community?

Before choosing a solver...

- What kind of problems do you want to solve?
- Do you/your client have a license to use it?
- How easy would it be to change?
- How much of the solver's API do you need?
- Do you really need a *specific* solver?

Helpful link: Decision-tree for optimization software

Modelling interfaces go beyond solvers

Sometimes you just want to instantiate a model and solve it, and which solver you use doesn't (really) matter.

That's what modelling languages are for

- :) Focus on the modelling, simpler syntax, solver-agnostic
- :| You may incur some performance cost
- :(You may not have access to all a solver's API (e.g. callbacks)

Many options:

Open-source: CMPL, CVX, JuMP, PyOmo, YALMIP, etc...

Commercial: AMPL, GAMS, AIMMS

Base rule

Always separate

generic code that can be re-used, from

specific code that only makes sense for a given application

and use an `import` to use the generic code.

Why so?

It is easier to navigate

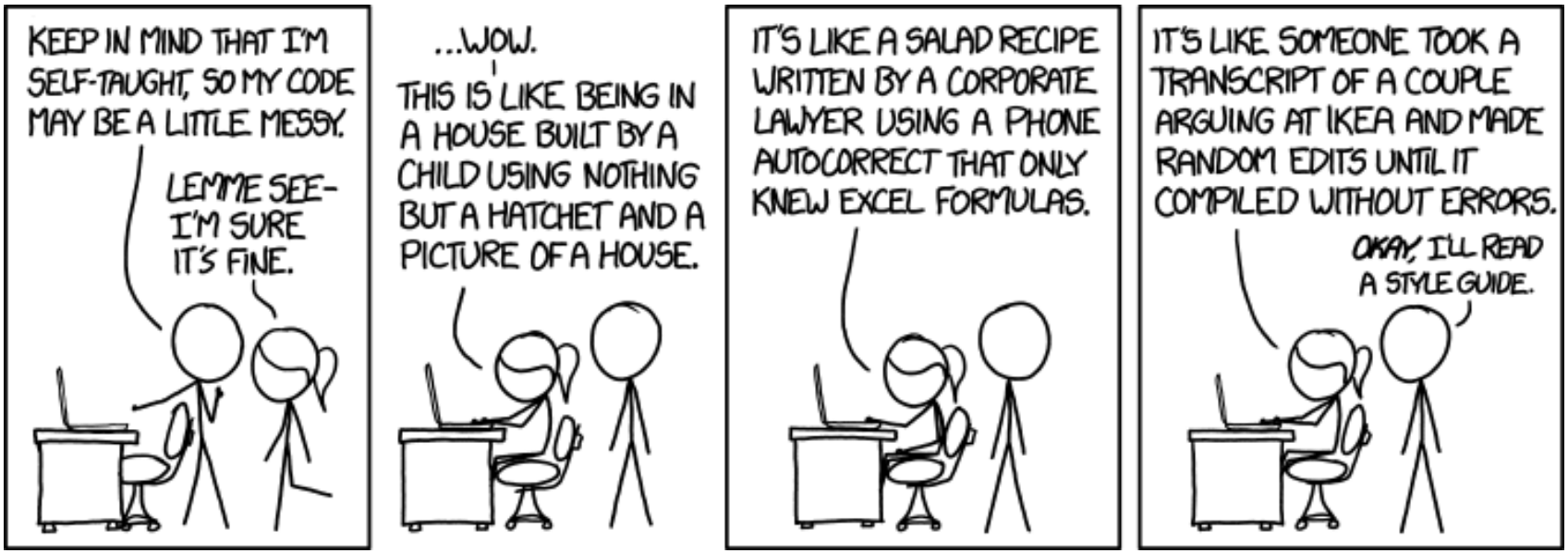
It allows you to modify one without having to change the other

Someone (you included) may want to re-use your code later

Typical repository structure:

```
--dat/                  # small data files [optional]
    instance.mps
--doc/                  # documentation
    cholesky.md
    algo.pdf
--examples/             # illustrative examples
--src/                  # source code (classes and functions)
    --Module1/
    --Module2/
    some_code.jl
--test/                 # unit tests
    runtests.jl
LICENSE                 # code license
README                  # short project description
```

Write code that other people want to read



How would you write an MILP?

$$\begin{aligned} \min_{B} \quad & \mathcal{X}^T B \\ s.t. \quad & xB = \mathcal{A}, B \in c \end{aligned} \qquad \text{or} \qquad \begin{aligned} \min_{x} \quad & c^T x \\ s.t. \quad & Ax = b \\ & x \in \mathcal{X} \end{aligned}$$

We all use *style conventions*, e.g.:

- x is the variable, c is the objective
- Upper-case denotes matrix, lower-case denotes scalar or vector
- ...

Same applies to code!

Style guide: *"a set of conventions (sometimes arbitrary) about how to write code for that project. It is much easier to understand a large codebase when all the code in it is in a consistent style."* - Google style guide

Style guides do not make your faster. They make it look nice.

A code with no style guide is like a paper without formatting: nobody wants to read it.

Some style guides:

- Python
  - (mandatory) PEP8, PEP257
  - Google Python style guide
- C++
  - Google C++ style guide
- Julia
  - Julia style guide

A useful tool (for Python): PyLint

Pick one and stick to it!

Version control

What is version control?

Why should I use it?

How do I use it?

What is version control?

Tracks the evolution of the code

Ensures different people collaborate without conflicting

Who uses it?

Everyone. And so should you!

See `version_control` notebook

Further resources:

- https://www.atlassian.com/git/tutorials
- https://try.github.io/
- https://github.com/ds4dm/tipsntricks/tree/master/git
- https://openclassrooms.com/en/courses/3321726-manage-your-code-with-git-and-github

Share your code!

Describing an **Algorithm** without **Implementation** is like stating a **Theorem** without **Proof**

#just_a_computational_conjecture

- Matteo Fischetti, ISMP2018

Several platforms (free plans available):

- BitBucket: `https://bitbucket.org/`
- GitHub: `https://github.com/`
- GitLab: `https://about.gitlab.com/`

What every paper should have:

Source code, scripts and data for running the experiments are publicly available at [...]

Unit testing

What is unit testing?

Why you should use it

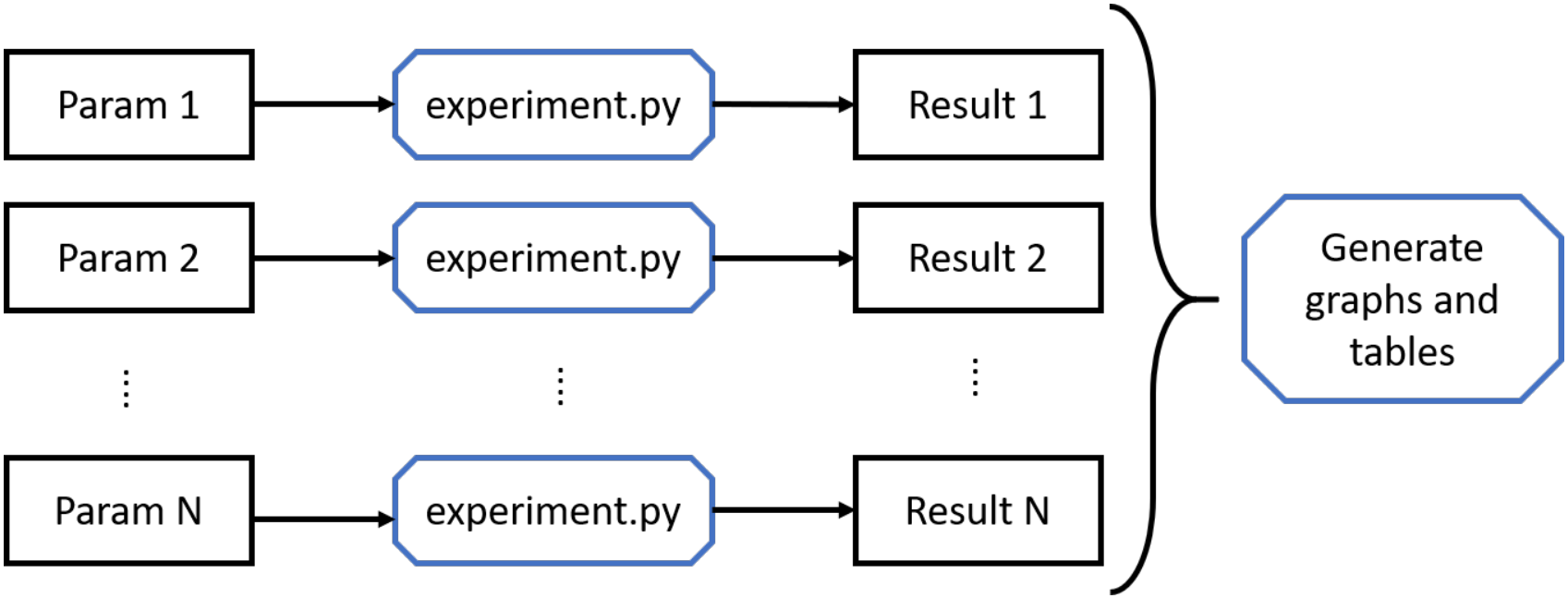
How to use it in practice

Demo

Unit tests in practice

1. Always write tests for your source code!
2. Run tests locally before making a commit
3. Continuous integration
   - Automatically run tests when modifications are pushed
   - Free for open-source projects
4. Unit tests do not prevent bugs (but they help)!
5. Most useful at later stages of your project (prevent breaks)

Numerical experiments typically look like



The "atomic experiment" (a.k.a, a job)

```
$ python experiment.py --arg1 p1 --arg2 p2 > output_p1_p2.txt
```

run `experiment.py`
with arguments `arg1=p1` and `arg2=p2`
and redirect output to `output_p1_p2.txt`

Examples:

- Process a given data file
- Read an instance from a file and solve it
- Train a ML model with given hyperparameters

Example workflow for running experiments

1. Generate the list of jobs

   ```
   python test_prepross.py > jobs.txt
   ```

2. Launch jobs

   ```
   cat jobs.txt | parallel -j 4
   ```

   (remember to re-launch if failures)

3. Generate graphs and tables

   ```
   python test_postpross.py
   ```

Sanity checks

- Use random seeds and save them (either as parameter or output)
- Check that you do output the data you need to output
- Do not run the same job twice
- Generate graphs/tables without having to re-run all jobs
- Watch out for disk space (data files, large outputs)
- Don't run more jobs than you have cores
- Check that everything runs as intended
- Ensure you can map results back to a job's parameters!

Wrapping-up:

- Use tools you're comfortable with
- Write code that other people want to read
- Use version control
- Seriously, use version control
- Get into the habit of unit tests
- Optimize your code intelligently
- Automate most of your experiments

Share your code! ;)

Thanks! Questions?