BONNES PRATIQUES DE DÉVELOPPEMENT LOGICIEL

Jeremy Laforet jeremy.laforet@utc.fr

Équipe NSE - D2.16 - 4372

PLAN

- Bonnes Pratiques
- Gestion de versions
- Tests
- Profiling
- Optimisation de code

BONNES PRATIQUES

Repris de software-carpentry.org

RULE 1

WRITE PROGRAMS FOR PEOPLE, NOT COMPUTERS

- Hard to tell if code that's difficult to understand is doing what it's supposed to
- Hard for other scientists to re-use it...
- ...including your future self

RULE 1A

A program should not require its readers to hold more than a handful of facts in memory at once.

- Short-term memory can hold 7±2 items
- So break programs into short, readable functions, each taking only a few parameters

RULE 1B

Make names consistent, distinctive, and meaningful.

- `p` doesn't help the reader's short term memory as much as `pressure`
- Don't use `temp` for both "temporary" and "temperature"
- `i`, `j` are OK for indices in small scopes

RULE 1C

Make code style and formatting consistent.

- *Which* rules don't matter — *having* rules does
- Brain assumes all differences are significant
- Every inconsistency slows comprehension

RULE 2

LET THE COMPUTER DO THE WORK

- Computers exist to repeat things quickly
- 99% accuracy ⇒ 63% of at least one error per hundred repetitions

RULE 2A

Make the computer repeat tasks.

- Write little programs for everything
- Even if they're called scripts, macros, or aliases
- Easier to do this with text-based programming systems than with GUIs

RULE 2B

Save recent commands in a file for re-use.

- Most text-based interfaces do this automatically
  - Repeat recent operations using `history`
  - "Reproducibility in the small"
- Saving history supports "reproducibility in the large"
  - An accurate record of how a result was produced
  - *If* everything can be captured

RULE 2C

Use a build tool to automate workflows.

- Originally developed for compiling programs
- Can be used whenever some files depend on others
- Makes workflow explicit

RULE 3

MAKE INCREMENTAL CHANGES

- Most scientists don't have "requirements"
    - They are their own users
    - Code evolves in tandem with research
- Closest fit from industry is *agile development*

RULE 3A

Work in small steps with frequent feedback
and course correction.

- People can concentrate for 45-90 minutes without a break
- So size each burst of work to fit that
- Longer cycle should be a week or two

RULE 3B

Use a version control system.

- Tracks changes
- Allows them to be undone
- Supports independent parallel development
- Essential for collaboration collaboration

RULE 3C

Put everything that has been created manually in version control.

- Not just software: papers, raw images, ...
  - Not gigabytes...
  - ...but metadata *about* those gigabytes
- Leave out things generated by the computer
  - Use build tools to reproduce those instead
  - Unless they take a very long time to create

RULE 4

DON'T REPEAT YOURSELF (OR OTHERS)

- Anything repeated in two or more places will eventually be wrong in at least one
- If it's faster to re-create than to discover or understand, *fix it*

RULE 4A

Every piece of data must have
a single authoritative representation in the system.

- Define constants exactly once
- Ditto file formats, geographical locations, ...

RULE 4B

Modularize code rather than copying and pasting.

- Reducing code cloning reduces error rates
- Cuts the amount of testing needed
- And increases comprehension

RULE 4C

Re-use code instead of rewriting it.

- It takes experts years to build high-quality numerical or statistical software
- Your time is better spent doing science on top of that

RULE 5

PLAN FOR MISTAKES

- No single practice catches everything
- So practice *defense in depth*

Note: improving quality increases productivity

RULE 5A

Add assertions to programs to check their operation.

- "This must be true here or there is an error"
- Like diagnostic circuits in hardware
- No point proceeding if the program is broken...
- ...and they serve as *executable documentation*

RULE 5B

Use an off-the-shelf unit testing library.

- Manages setup, execution, and reporting
- Re-run unit tests after every change to the code to check for *regression*

TESTING IS HARD

- "If I knew what the right answer was, I'd have published by now."
- Compare to experimental data
- Or to analytic solutions of simple problems
- Or to old (trusted) programs
- If nothing else, forces scientists to document what "errors" are acceptable

RULE 5C

Turn bugs into test cases.

- Write a test that fails when the bug is present
- Then work on the code until that test passes...
- ...and no others are failing

TEST-DRIVEN DEVELOPMENT

- Why wait? Always write the tests, then the code
- Improves focus
- Encourages writing testable code
- And ensures tests actually get written...
- "Red, green, refactor"

RULE 5D

Use a symbolic debugger.

- Explore the program as it runs
- Better than print statements
  - You don't have to re-run...
  - ...or guess in advance what you'll need to know
- Use *breakpoints* to stop program at particular points or when particular things are true

RULE 6

OPTIMIZE SOFTWARE ONLY AFTER IT WORKS CORRECTLY

- Even experts find it hard to predict performance bottlenecks
- Small changes to code often have dramatic impact on performance
- So get it right, *then* make it fast

RULE 6A

Use a profiler to identify bottlenecks.

- Reports how much time is spent on each line of code
- Re-check on new computers or when switching libraries
- Summarize across unit tests

RULE 6B

Write code in the highest-level language possible.

- People write the same number of lines of code per hour regardless of language
- So use the most expressive language available to get the "right" version...
- ...then rewrite core pieces (possibly in a lower-level language) to get the "fast" version

RULE 7

DOCUMENT DESIGN AND PURPOSE, NOT MECHANICS

- Goal is to make the next person's life easier
- Focus on things the code *doesn't* say
  - Or doesn't say clearly
  - E.g., file formats
- An example is worth a thousand words...

RULE 7A

Document interfaces and reasons,
not implementations.

- Interfaces and reasons change more slowly than implementation details, so documenting them is better economics
- And most people care about using code more than understanding it

RULE 7B

Refactor code in preference to explaining how it works.

- Good code can be understood when read aloud
- Good programmers build libraries so that solving their problem is straightforward
- Again, "red, green, refactor"

RULE 7C

Embed the documentation for a piece of software
in that software.

- Specially-formatted comments or strings
- More likely to be kept up to date
- More accessible to interactive help
- Many modern tools embed code in documentation rather than vice versa

RULE 8

COLLABORATE

- Computers were invented to calculate
- The web was invented to collaborate
- Science is more fun when it's shared

RULE 8A

Use pre-merge code reviews.

- Have someone else review changes *before* merging in version control
- Significantly reduces errors
- Good way to share knowledge
- It's what makes open source possible

RULE 8B

Use pair programming
when bringing someone new up to speed
and when tackling particularly tricky problems.

- Two people, one keyboard, one screen
- An extreme form of code review
- Can get a bit tired if done all the time…

RULE 8C

Use an issue tracking tool.

- A shared to-do list
    - Items can be assigned to people
    - Supports comments, links to code and papers, etc.
- "Version control is where we've been, the issue tracker is where we're going"

GESTION DE VERSIONS

- Archivage des modifications incrémentales
- Identification des auteurs
- Possibilité d'avoir des branches parallèles

Il existe de nombreux systèmes, un des plus courant aujourd'hui: GIT

GIT

- Décentralisé
- Dépot local (+ distant)
- Hébergement personnel ou forges (github, bitbucket…)

TESTS

test-driven development

- Écrire les tests avant le code
- Puis écrire le code qui passe les tests
- Faire passer les tests à chaque itération du code

Les tests sont une formalisation du cahier des charges et permettent d'éviter les régressions ("ça marchait avant.")

EN PYTHON

- pytest
- unittest
- nose

DEMO: PYTEST

PROFILING

Analyser les performances du code en terme de mémoire ou de temps de calcul.

- Permet d'avoir une vision objective des goulets de performances
- Savoir où et quoi optimiser pour améliorer effectivement les performances

TYPE DE PROFILER

- cProfile: analyse du temps de calcul, à l'échelle de la fonction
- kernprof: analyse du temps de calcul, ligne par ligne
- memory_profile: analyse de la consomation mémoire, ligne par ligne
- guppy: analyse des types de donées utilisés

OPTIMISATION DE CODE

- Optimiser là où c'est nécessaire
- Vérifier l'absence de régressions

PISTES

Appliquer la technique la plus adapté au problème local

- Limite d'un construction particulière du langage: ré-écriture
- Nombreux traitement indépendants: parallélisation
- …

EXEMPLE

En python les boucles sont assez lentes, donc autant que possible les remplacer pour des fonctions dédiées

```
for x in array:
    sum+=x
```

```
numpy.sum(array)
```

CONCLUSION