Best practices in scientific programming
Software Carpentry, Part I

Pietro Berkes, Brandeis University

Outline

- Part I Agile development for scientists
  - Good programming practices
  - Test driven development, late optimization, refactoring
  - This part is not specific to any programming language
- Part II Python tools
  - Version control systems
  - Testing, debugging, profiling
  - Focus on standard, out-of-the box Python tools

Modern programming practices and science

- Many of us have to routinely write computer programs, but few of us have been trained to do so
- Good programming practices can make a lot of difference
- Development methodologies have been introduced for the development of commercial software, but we can learn *a lot* from them about making science

Scenarios

- Lone scientist, coding up a model or data analysis tool for a research project
- Small team of scientists, working on a common library
- The bottleneck is developing speed, not execution speed
- Need to try out different ideas: rapid prototyping, re-use code, identify common patterns and use known solutions

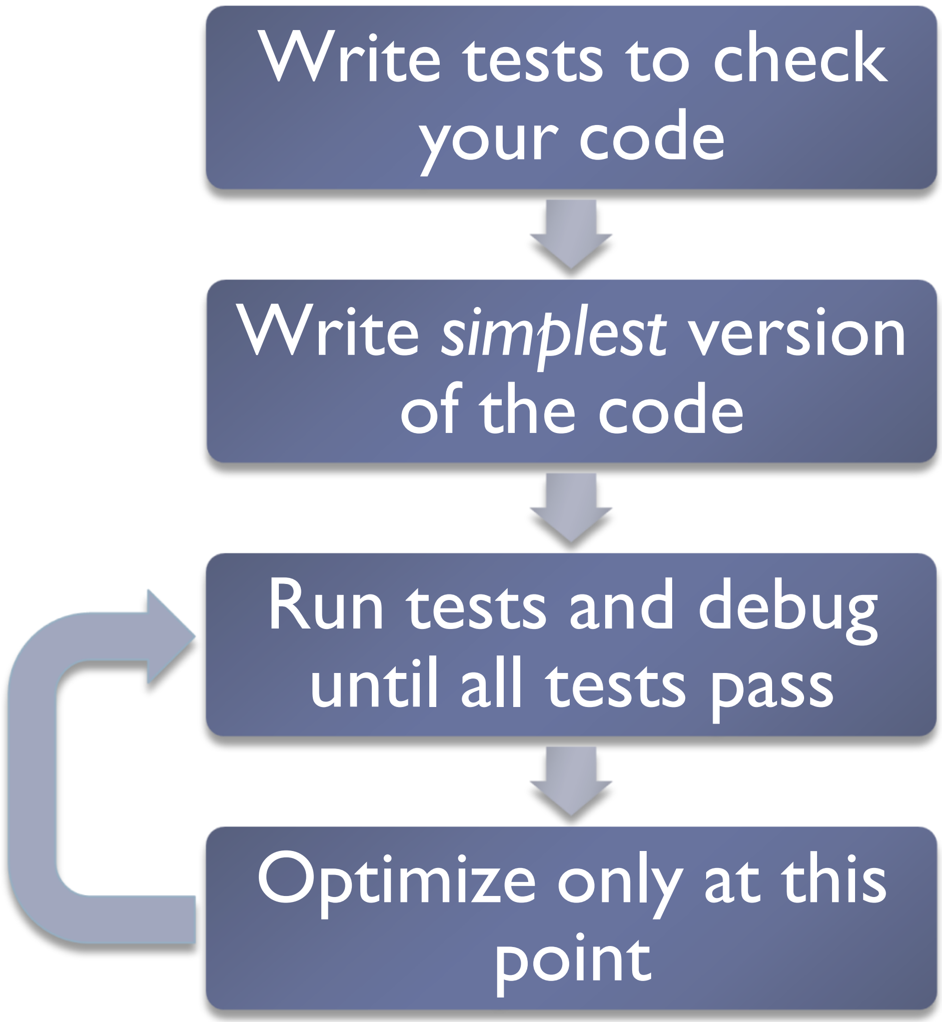
Requirements for scientific programming

- Every scientific result (especially if important) should be independently reproduced at least internally before publication (DFG, 1999)
  - Translation: there must be guarantees that the source code works as advertised (testing frameworks, pair programming)
- Increasing pressure for making the source code used in publications available online (especially for theoretical papers)
  - Translation: you shouldn't be embarrassed of publishing your code
  - Your code must be readable and easily reusable

Agile development

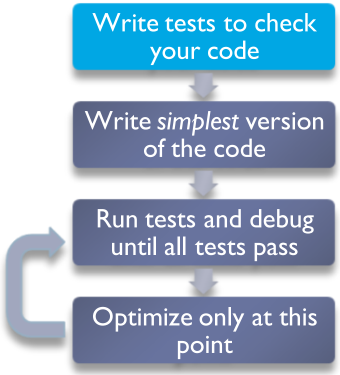
- Generic name for set of more specific paradigms, most influential eXtreme Programming (XP), formulated in the 90s by Kent Beck, Ward Cunningham, and Ron Jeffries
- Set of good programming practices, from design of software to development to maintenance
- Particularly suited for small teams (<10 people) facing unpredictable or rapidly changing requirements (sounds familiar?)

The basic agile development cycle



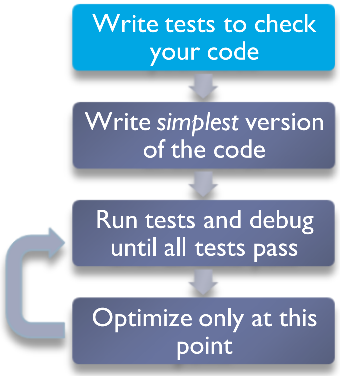
Planning

- Story-based planning
- Writing tests before actual code helps designing the interface
- Use *spike solutions* to test approaches
  (i.e., write a toy implementation / proof-of-concept)



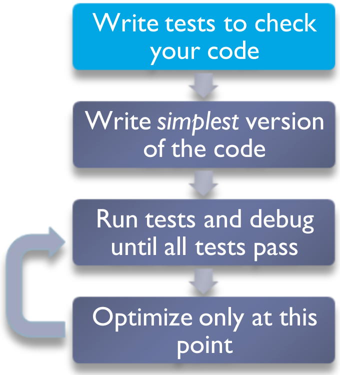
Test-driven development

- Tests are *crucial* for modern programming. Tests become part of the programming cycle and are *automated*
- Write test suite (collection of tests) in parallel with your code
- External software runs the tests and provides reports and statistics



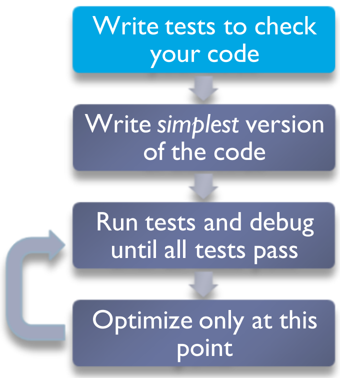
Testing benefits

- Encourages better code and optimization: code can change, and consistency is assured by tests
- Faster development:
  - Bugs are always pinpointed
  - Avoids starting all over again when fixing one part of the code causes a bug somewhere else
- Installation check for users if you plan to distribute your code
- To reviewers: "I know my code works, because it passes these tests"



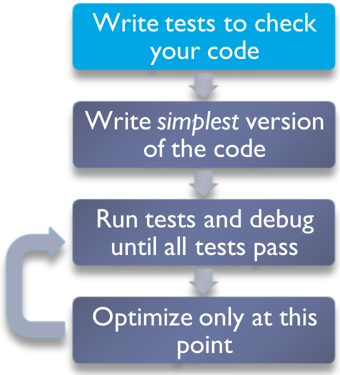
What to test and how

- Test with simple (but general) cases, using hard coded solutions
  - E.g., test that `sum(2, 3) = 5`
- Test general routines with specific ones
  - E.g., test `polyomial_expansion(data, degree)` with `quadratic_expansion(data)`
- Test special or boundary cases
  - E.g., test `has_prefix(string, pfx)` for `pfx=""`
- Test that the code raises meaningful errors when wrong data is passed
  - Relevant when writing scientific libraries



Example: eigenvector decomposition

- Consider the function `values, vectors = eigen(matrix)`
- Test with simple but general cases:
  - use full matrices for which you know the exact solution (from a table or computed by hand)
- Test general routine with specific ones:
  - use the analytical solution for 2x2 matrices
- Test with boundary cases:
  - test with diagonal matrix (is the algorithm stable?)
  - test with a singular matrix



"I'm writing a learning algorithm / probabilistic algorithm. How can I possibly test it?"

Turn your validation cases into tests

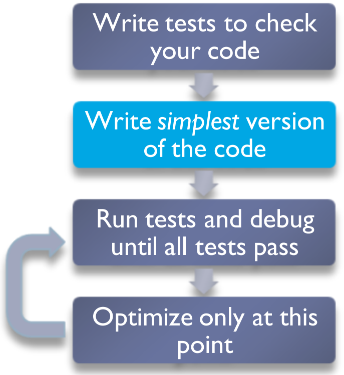
Think of simple, artificial cases

Probabilistic becomes deterministic with lots of data or disappearing noise

Test all sub-parts of the algorithm

Start simple

- Write small, testable chunks of code
  - Write intention-revealing code
  - Separate testable parts from main application
- Do not implement a general problem-solving framework for a specific problem
  - unnecessary features are not used but need to be tested and maintained
- Do not try to write complex, efficient code at this point



“The whole point of my research is implementing an efficient algorithm…”

The *algorithm* should be efficient, not its first implementation

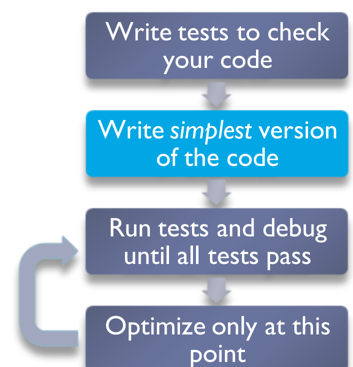
Do not use optimization tricks

Re-use standard data types

Do not vectorize if a for-loop will do

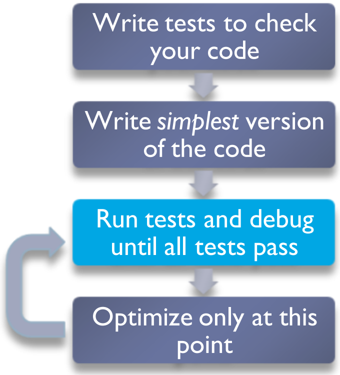
Collaborating

- Pair programming
  - One programmer sits at the computer and does the coding, the other looks and keeps an eye on the big picture
  - Pairs are fluid
- Code must be formatted to agreed coding standards (Python: PEP8)
- Write intention-revealing code
  (comments should be mostly not necessary)
- Keep your code documented (docstrings!)
- Use Version Control Systems to handle shared code



How to handle bugs

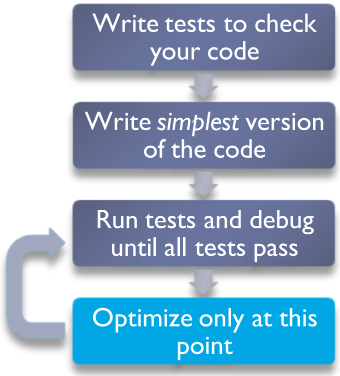
1. Isolate the bug
   - Test cases should already eliminate most possible causes
   - Use a debugger, not `print` statements
2. Add a test that reproduces the bug to your test suite
3. Solve the bug
4. Run *all* tests and check that they pass



Optimization

- Usually, a small percentage of your code takes up most of the time

1. Identify time-consuming parts of the code (use a profiler)
2. Only optimize those parts of the code
3. Keep running the tests to make sure that code is not broken



“When should I stop optimizing?”

As soon as it’s fast enough

When all the obvious optimizations are implemented

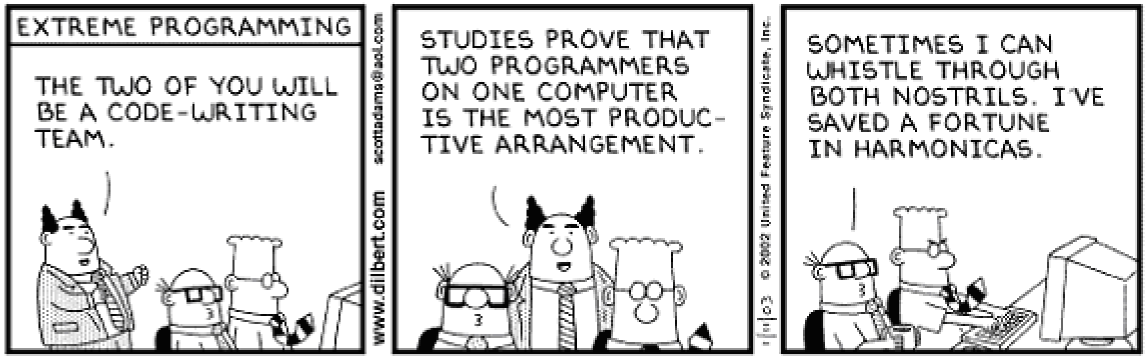
Consider also alternative forms of optimization: running remotely on a faster machine, having multiple runs in parallel, ...

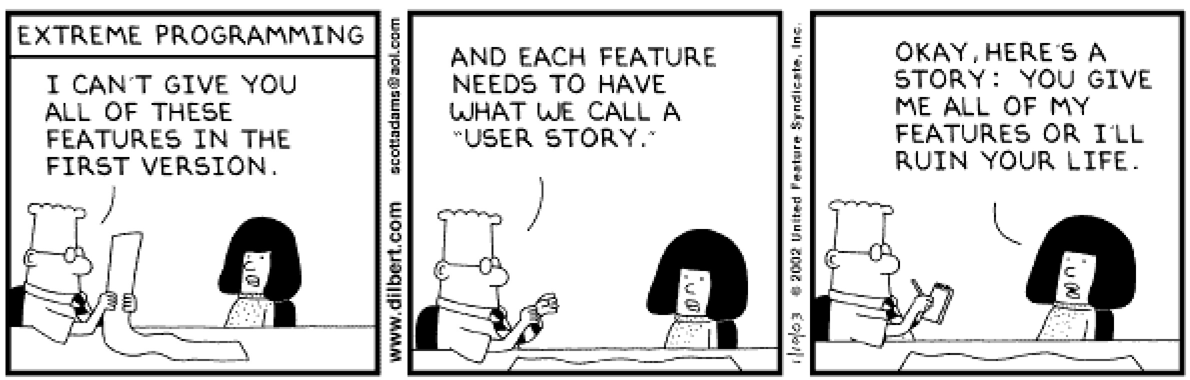
Adding new functionality

1. Implement new functionalities as in basic development cycle
2. Refactor the code
   - Re-organize your code without changing its function
   - E.g., remove duplicated code, break down complex functions in simpler parts, rename variables and functions to make intention clearer
   - Series of recipes for these common operations: how to do them without breaking the code
   - Do it in small steps, keep testing
   - Many modern IDEs have refactoring tools

Final thoughts on Part I

- **Do not underestimate these practices**
- Adapt these techniques to your needs, but try to keep in mind the basic principles
- Never stop testing


EXTREME PROGRAMMING
THE TWO OF YOU WILL BE A CODE-WRITING TEAM.
STUDIES PROVE THAT TWO PROGRAMMERS ON ONE COMPUTER IS THE MOST PRODUCTIVE ARRANGEMENT.
SOMETIMES I CAN WHISTLE THROUGH BOTH NOSTRILS. I'VE SAVED A FORTUNE IN HARMONICAS.


EXTREME PROGRAMMING
I CAN'T GIVE YOU ALL OF THESE FEATURES IN THE FIRST VERSION.
AND EACH FEATURE NEEDS TO HAVE WHAT WE CALL A "USER STORY."
OKAY, HERE'S A STORY: YOU GIVE ME ALL OF MY FEATURES OR I'LL RUIN YOUR LIFE.