Best Practices in Software Development and Design

Martin Kümmel

References

- Nothing in the library (Sect. Nc, Nd) :-(
  - Ask **Harald Lesch** if you need something!
- On the internet:
  - Google style guide for python: https://google.github.io/styleguide/pyguide.html
  - Google style guide for C++: https://google.github.io/styleguide/cppguide.html
- Books (in my shelf):
  - McConnell: Code Complete (Developer Best Practices)
  - Pressman: Software Engineering: A Practitioner's Approach
  - Gamma, Helm, Johnson, Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software

Some motivational statements

- Software Development is a young science!
- Software Development has **no** absolute rules!
- Software Development is a key asset in science/astronomy!
- Lots of free SW available → you can concentrate on science code!!
- Everyone writing code needs knowledge on Software Development!
- Learning proper Software Development is **not** a waste of time!
- Learning proper Software Development does not take much time and effort!

Not all software is equal!

- Software to solve my singular problem.
- Software to solve a variety of problems.
- Software to be published (paper or github).
- Software to be shared (within the group or collaboration).
- Software to contribute to a large package or project.

- Growing need for a more formal approach;
- Growing need for Software Engineering and Design;

Problems in the example code

- “Old” comments are still in;
- Program is rather large (136 lines);
- Imports are not ordered;
- Some very long lines (>200chars);
- No modularization (almost);
- Inline function using 'global' variables (disaster recipe);
- Few inline comments;
- Different style in existing inline comments;
- No proper parameterization;

Problems in the example code (cont.)

- Unnecessary (dubious?) deletions;
- No naming scheme;
- Important formulas not isolated;
- Important formulas not commented, referenced;

Best Practices

- Structure all code!
- Use naming schemes (`smallCamelNames` or `many_underscored_names`)!
- Limit the size of code units (funtions, methods)! Set yourself a goal, e.g. 200 code lines!
- Create logical code units:
  - Main
  - In/output
  - Several units with computations/business logic
- Do **not** overload the interfaces (5 plus/minus 2 parameters)
- Code loosely:
  - 1/3 logical code
  - 1/3 comments
  - 1/3 separators
- Use argument parsers (argparse in python, boost in C++)
- Use standard formats for in/output:
  - FITS
  - ASCII
  - JSON
  - **No** pickles, numpy.save()

Use an Integrated Development Environment

- One tool for:
  - Code development;
  - Compiling;
  - Testing;
  - Running;
- Autocompletion (→ less typing, less spelling mistakes)
- Easy code editing (file-jogging, indentations, syntax colouring);
- Built in syntax checking:
  - Never undefined variables in python!
  - Some problem can “only” be found with an IDE;
- Top dog Eclipse:
  - Supports all languages;
  - Very high integration is possible;
  - ... but there are many others;

Use a versioning tools

- To answer typical questions such as:
  - That worked yesterday, or?
  - How did I get this result a week ago?
- Adds the time dimension to your software!
- To preserve code snippets over time;
- Use for all software!
- Existing tools:
  - Git/github (see last talk)
  - svn
  - CVS

Generate Test Cases

- There are test cases at different levels:
  - Unit tests (at elementary function/method level);
  - End-to-End tests for entire programs;
  - System tests if not only software is involved;
- In an ideal world:
  - Test driven software development!
  - Make the tests covering all functionality first;
  - The development is finished once all tests pass;
- In the real world:
  - The functionality is rarely clear at the beginning;
  - Writing tests takes a lot of effort;
  - Often astronomy software and its testing needs lots of data;
- Test driven development is great if possible (“pure” software projects)

Software Design

- Strictly necessary in two cases:
  - For large projects (>10,000 code lines);
  - When using an Object Oriented programing language;
- Formal design approach usually not necessary;
- Problems:
  - Scope and extend of a software project not clear at the start (usually it “grows”)
  - When you realize that you need formal design, it is usually too late...
- Two step approach??:
  1. Make the software such that it works.
  2. Re-do the software with proper design.
- A missing design can limit software development.
- Use established approaches:
  - Bottom up;
  - Top down;
  - Uses cases;
  - Design patterns;
- Software design with `ctype`, `cython`, `Jupyter`???