MarSci Programming Standards

Jonas Hausruckinger, Marek Jiruse, Andreas Kersting, Guenter Pesch, Alexander Riemer, Frank Rosenthal, Barbara Wolf

Goals

- Make collaborations within the department easier
- Reduce the time required to understand code written by someone else
- Improve the general quality of code written in the department, especially in terms of software engineering

What these standards are meant to cover

- It should be focused on our department and its specific needs, and not be general programming standards
- Be language agnostic (as much as possible)
- Not be a list of best practices (for a specific language)
- Each project might have an own programming standard which can be more specific and / or expand on the general department standards.

Code Style

1) Follow the Style Guide

Every programming language has a style guide that tells you in great detail how to indent your code, where to put spaces and braces, how to name stuff, how to comment – all the good and bad practices.

Read the guide carefully, learn the basics by heart, apply the rules, and your code will be much easier to understand, handle, and modify for others as well as yourself.

Use tools which automatically verify that your code adheres to a specific style guide wherever possible.

Language-specifics:

- For R:
  - Hadley Wickham's style guide
  - Use RStudio's built-in capabilities to automatically analyze your code and whether it adheres to the basics of the aforementioned style guide: RStudio article.
- For Python:
  - The Hitchhiker's Guide to Python
  - PEP 8 – Style Guide for Python Code
  - When using Eclipse + PyDev for developing you can automatically check your code for adherence to the standards.

2) Use Descriptive Names

> "There are only two hard things in Computer Science: cache invalidation and naming things."
>
> – Phil Karlton

- Adhere to the naming rules outlined in the specific language's style guide.

- Use long descriptive names, like `removeZeroVariance`, to help, now and in the future, yourself as well as your colleagues to understand what the code does.
  - The only exception to this rule concerns the few key variables used within a function's body, such as a loop index, a parameter, an intermediate result, or a return value.
  - Use named parameters for functions unless it's obvious which parameter is used, e.g. `sample(x = cities, size = 10, replace = TRUE)` but `sum(a, b)`
- Think long and hard before you name something:
  - If you're naming a function, does the name describe what the function returns?
  - Is the name accurate?
    - Did you mean `highestPrice`, rather than `bestPrice`?
  - Is the name specific enough?
  - Does the name form match that of other similar names?
    - If you have a function `ReadEventLog`, you shouldn't name another `NetErrorLogRead`.
- Avoid Using Magic Numbers. Magic numbers are hard-coded constants in the code.
  - Alternatives to magic numbers:
    - Use as function parameter.
    - Define as a variable at a prominent place (e.g. beginning of the script).
    - Compute it in the code (24 * 7 instead of 168).
    - Use a generating function from a package, e.g. days in month.
  - The same rule applies to string constants, e.g. column and path names.
- Strictly define constants, which may never change over time, in one central place.

3) Comment and Document

- Write the comments as you develop the code; if you think you'll add them later, you're kidding yourself.
- Comments, documentation, git commits and everything else should only be written in English!
- Comments should ideally tell the reader **WHY** you are doing something, not **WHAT** you are doing at that moment.
- Make sure that your comments still fit the respective code section AFTER you have updated it.
- Summarize in a comment the role of each file and routine, and the major steps of complex code.
  - Give a rough indication of the running time for long-running code sections in your scripts.
- Start all of your routines you write, e.g. a function, with a comment outlining what the routine does, its parameters, and what it returns, as well as possible errors and exceptions.
  - Ideally, use automatic generation of documentation, e.g. Roxygen for R.

  - This can be skipped for small helper routines whose functionality is self-apparent.
- In addition, ensure that your code as a whole (for example, an application or library) comes with at least a guide explaining what it does; indicating its dependencies; and providing instructions on building, testing, installation, and use. This document should be short and concise; a single README file is often enough.
- Use 'TODO' comments.
- Explain why certain sections are (temporarily) 'commented out'. In general, try to keep this practice at a minimum and keep track of changes with Git instead.

Software Engineering

4) Don't Repeat Yourself (DRY)

> 'When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements. Additionally, elements that are logically related all change predictably and uniformly, and are thus kept in sync.'

– Wikipedia

- If you are using a certain piece of code more than once think about putting its functionality in a function to avoid ‘repeating yourself’. Every instance of duplicated code increase its maintenance costs!
- Exceptions are possible for small adhoc analyses and scripts, but the longer your expected usage of the code the more good design pays off in the long run.

5) Split Your Code into Short, Focused Units

- Each routine, e.g. function, should concern one single thing. If a code unit undertakes diverse responsibilities, consider spliting it accordingly.
- Every function, or logical code block should have a resonable length (max. about 100 rows excluding the initial documentation). If it’s longer, consider spliting it into shorter pieces.
  - Avoid too many nested levels. Three levels should be sufficient in almost all cases.
- Even within a routine, divide long code sequences into blocks whose function you can describe with a comment at the beginning of each block.
- Similarly, files / scripts should ideally also only address a limited scope:
  - One file for a longer function / one file for several, shorter but related functions.
  - One file for central constants / global variables. Try to avoid using global variables as much as possible.
  - One file for the overall orchestration.
- If possible, group thematically related functionality in a module, package, etc. (e.g., data preprocessing, modeling, output generation). This gives a better overview of your code and enables the reuse of specific functionalities in other projects.

6) Don’t Overdesign

- Keep your design focused on today’s needs. Your code can be general to accommodate future evolution, but only if that doesn’t make it more complex.
  - Don’t spend a lot of time on implementing functionality you think **might** be needed in the future. Often it is never needed.
- On the other hand, when the code’s structure no longer fits the task at hand, don’t shy away from refactoring (changing the code structure without changing its functionality) it to a more appropriate design. Again, make sure to also update your comments accordingly.

7) Be Consistent

- Do similar things in similar ways. If you’re developing a routine whose functionality resembles that of an existing routine, use a similar name, the same parameter order, and a comparable structure for the code body.
- If you are working on legacy code, keep using that code’s (implicit) style and logic. So avoid re-writing smaller parts of an existing code base, simply to adhere to a certain outside standard. Internal consistency is important!

8) Avoid Premature Optimization

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%“

– Donald Knuth, 1974

- Optimization or performance tuning should generally be performed at the **end** of the development stage. Premature optimization can often result in a design that is not as clean as it could have been, because the code is complicated by the optimization.
  - Additionally, performance bottlenecks might only become apparent after the design is fully implemented.
  - Nevertheless, make sure to use the right tools / modules / functions from the start of your development process.
- Before optimizing a certain part of a program, make sure to use profiling / benchmark tools to identify which specific part should be optimized.
- Make sure the optimization is really needed: If one hour is spent to speed up a program from 1 second to 0.01 seconds the program needs to run several thousand times to at least redeem the time invested for the optimization.

Best Practices

9) Use Framework APIs and Third-Party Libraries

- Learn what functionality is available through an API in your programming framework, and also what's commonly available through mature, widely adopted third-party libraries.
- Libraries supported by your system's package manager are often a good bet to achieve results in a more efficient time than writing your own, original code.
- Use that code, resisting the temptation to **reinvent the wheel**. Examples:
  - String processing (upper case / lower case, counting, splitting, etc.).
  - Searching, sorting, etc. routines.
  - Parsing routines (JSON, XML, etc.).
  - Inventing regular expressions for everyday problems (e.g. URL validation).
  - Statistical methods, data wrangling.
  - Date / time computations.
  - Format conversions.
  - Logging and error handling conventions.
- Make sure to use identical versions of third-party packages within a project team.

10) Put Everything Under Version Control

- All elements of your system should be under a professional version control system like Git:
  - Code
  - Documentation
  - Data for testing code (< 1 MB)
- COMMIT best practice:
  - when:
    - if a new feature is finished or a bug solved
    - after you have tested your code
  - how:
    - not two different changes as one commit
    - always use useful commit messages, e.g. not 'update from xyz'
    - Tag certain (e.g. stable) versions in history with descriptive labels
    - Use '.gitignore' to exclude specific files from version control

11) Test (your) Code Thoroughly

- Ideally, let someone else (with the appropiate background) test your own code.

    - If no one is available, make sure to thoroughly test your own code.
- Make sure to test (possible / probable) edge cases
- Use automated tests (where applicable and sensible)
    - For example, unit test packages like testthat (for R) or pytest (for Python)
    - Write (unit) tests if you want to be confident that your (short, focused) routine still performs its intended functionality after modifications.
    - After fixing a bug, write an automated test to catch the same bug in the future.
    - Try to keep the runtime of tests as low as possible.
    - This approach simplifies debugging by allowing you to catch errors early, close to their source.
- Don't spend too much resources to test non-critcal functionality, but make sure to properly test critical functionality intended for deployment.
- Specifiy best practices for error handling within your project team.

12) Check for Errors and Respond to Them & Practice Defensive Programming

- Note unexpected / dangerous behavior within your comments.
- Thorough error / exception handling
    - Do not ignore errors!
    - Use helpful, descriptive error / exception messages.
    - Shuting down the programm in a controlled way is also a valid way of handling (unrecoverable) errors.
- Try to 'fail fast' - your code should raise an error as soon as something goes wrong. Check your relevant assertions as soon as possible in your routine, e.g. before a lengthy computation not afterwards.
- If sensible, use logging (from existing frameworks) and monitor your KPIs.

Applying these Code Standards

Knowing When and How to Deviate from These Standards

- Try to stick to these standards as much as possible. Obviously, always use common sense! While small deviations might not seem very harmful by themselves, they accumulate over time and can become big headaches.

Study Best Practices and Sources of Good Code

- Learn from best practices for the language you are working with as well as the domain-specific best practices and apply them where and when possible.
- Look at the source code of popular and widely-used packages to see how the top of their field writes their code.