CRESYM Software Development Guidelines

In general, any software must be delivered with the aim of clarity, modularity, completeness, interoperability, maintainability, and standardisation. This document intends to provide guidelines to ensure a minimum quality level and harmonised project results. It will be updated as the subject matures in CRESYM and with partners consultation.

These guidelines are not absolute requirements. However, they should constitute a reference for main software development choices.

Being aware that some guidelines might require more coding experience, a first training session has been provided with CodeRefinery in September 2024, in line with these guidelines. All training material is available here: https://coderefinery.github.io/reproducible-python/. More sessions will be programmed.

1) Cleanness (required)

- **Clear naming:** all variable names should make sense, are specific, in English and respect the standard of the programming language.
- **No hardcoding** (use of config files), changeable values should never be hardcoded, main parameters can be configured from the command line/config file. Examples: all paths for inputs/outputs, reference files and directories, filtering parameters, random number generation seeds, variant algorithms, verbose *etc*.
- **Modularity** (function, objects): the "Do not Repeat Yourself" rule is applied (functions should not be longer than a screen, functions have only one function). Pure functions[1] are always preferred. "Project code" is differentiated from "Library code" (for instance: data handling functions).
- **Reuse of existing robust libraries, package managers** (*with pip for instance*).
- **PEP-8 convention for Python (**https://peps.python.org/pep-0008/**), C++ core guidelines for C++ (**C++ Core Guidelines (isocpp.github.io)**), Oracle code convention (for Java)** codeconventions-150003.pdf (oracle.com), etc.
- **Refactoring**: Regularly, at least before the first uploading on GitHub; As soon as debugging becomes too heavy; Every major version. "Code smells" (duplicated code, long functions, long parameter lists) are rectified. First focus on small changes (giving variables better names, making global variables local to a function, eliminating obvious duplicate code, breaking up large functions into smaller ones, grouping related functions into classes, and reorganizing functions and classes into appropriate files). *Error correction libraries can be downloaded in most IDE.*

[1] Pure functions have no notion of state: They take input values and return values. Given the same input, a pure function always returns the same value.

Extra source: https://arxiv.org/pdf/1210.0530.pdf

Additionally, tools such as SonarQube, Azure, Jenkins can be used to provide a full code analysis.

2) Documentation

Required (progressively while TRL increases)

- **(Efficient) In-code documentation:** The code is "self-documented" including docstrings, comments and clear structure and naming. Comments explain "why" the code is written this way, while what it does should be explicit from the naming and code structure (e.g. by imitating human sentences: function=verb, variable=name, "is" for Boolean *etc.*). For automatic documentation, see below.
- **License file** (open source, without copyleft[2]): Apache License, Version 2.0 or Mozilla Public License 2.0 are selected for new software.
- **(Small) Examples**: Examples are included to illustrate the tool's functions and to help any external users to understand it. They are included in the testing procedure and in the documentation of the functions if any. There are never too many examples, they should show the main functionalities of the software.
- **README file**: Written as if this was the only thing users will read, include: the code purpose, how to install and configure, where to find the full documentation, the license, how to test it and acknowledgements (including recommended citation), and a QuickStart guide. The README must be in .txt, .md, or .html.
- **Requirements/environment file** is included, listing all dependencies needed. The project can be built under a virtual environment (*e.g. using Conda or Anaconda in Python*) to clarify dependencies formally and ensure its reusability.
- **Installation instructions, User Manual,** a quick start guide: Can be based on text, pictures, class diagrams, videos, GIFs, code to copy-paste *etc.*
- **Citation file (CCF), associate a DOI (Zenodo/Figshare)**: It can also be written in the README (as a BibTeX entry for instance), and a written reference for your publication in your README. Free way to get a DOI: submitting to the Journal of Open-Source Software.

Advised

- **Tutorial/demo:** with a video for instance.
- **API documentation**: with at least typed inputs and outputs, the type of errors it can raise. Objects should have their methods and attributes described. *Style guides exist (google.github.io/styleguide), and tools to automatically generate the doc (github.com/sphinx-contrib/napoleon*).
- **Automatic documentation** (*Examples of tools: Doxygen, MkDocs, Sphinx etc.*)

[2] CRESYM has elected a non copyleft license, for its partners to be free to reuse the code internally.

- **Version controlled documentation**: To be kept inside the Git repository. The documentation should include the main changes between the two versions. *Examples of tools to generate such documentation: readthedocs.org, zenodo.org.*
- **Help command, version number:** For command line interfaces (usage, subcommands, options/arguments, environment variables, examples etc.), provide an –help command for the user. *Tools exist to help building any command (for Python for instance: click.pocoo.org)*
- **Error messages** that provide solutions or point to the documentation (they state what the error is, what the state of the software was when it generated the error, and either how to fix it or where to find information relevant to fixing it). Where possible, use the log classes and their levels (fatal, error, warning, info, debug, trace).

3) Tests

Required (progressively while TRL increases)

- **Functional and integration test cases:** should have a minimal code coverage value (e.g. 75%). Include various test cases to evaluate the performance of the code.
- **Unit tests**: Tests of individual functions. *Can be included with the help of dedicated modules (e.g. "unittest" in Python: https://docs.python.org/3/library/unittest.html, Pytest). In Github, the tool Codecov can be used to see if tests sufficiently cover the code base.*
- **Non-Regression Tests**: to be run before committing modifications, and progressively enriched with the functional/integration test cases created for each new function. Can be a mix of unit, functional and examples. Can be added to GitHub Actions to run the tests automatically on every change.
- **Documentation**: Tests should be well documented (What is tested? How is it tested?), and easy to launch (*e.g. with a –test command*).
- **Testing the software compilation and installation on the expected environment** (OS, third parties and environment variables). Both release and Debug mode should be tested. *Physical or virtual machines, or containers like Docker can be used.*
- **Testing the software with new users**

Advised

- The "test-first" approach is applied: the test is written before the function itself (to identify wrong results and not only bugs).
- Provide a workflow that automatically sets the software environment, compiles and runs the tests of the software within it.

4) Versioning

Required

- **Git**: As a first step, Cresym's Github private repository or other repository can be used for software versioning (for harmonization purposes, GitHub is preferred). Once the code is ready for publication, the project is moved to a public repository. Permissions are clearly defined in both steps (who can access, modify, ask for a pull request *etc.*). Use Issues and Pull Requests (specific templates can be used). Follow advice on commits. For collaborative work, main project members use clones and branches for developing new functions (or any working line), other contributors fork the original repository to their workspace (before cloning and branching). See also advice for avoiding and solving potential conflicts.
- **Early and regular releases,** commits should include a limited number of modifications.
- **Semantic versioning** (semver.org) should be used ("1.0.1" = "MAJOR.MINOR.PATCH"):

1. MAJOR version when you make incompatible API changes
2. MINOR version when you add functionality in a backward compatible manner
3. PATCH version when you make backward compatible bug fixes

Tags can also be used (*using shields.io for instance)*.

5) Other

Loosely Required

- **Language: Python** (for scientific programming prototypes)
- **Reproducibility**: Releases are versioned, with a version for each published paper. Reproducible workflow (Shell, Jupiter, Snakemake) is used. The version number is available from the command line ('-version' or '-v'). The code produces identical results when given identical inputs (if randomness is used: the seed is set to a consistent value). The project is built under an isolated, virtual environment (*e.g. using Conda or Anaconda in Python*), associated with file listing dependencies (*e.g. requirements.txt, environment.yml in Python*).
- **Interoperability:** must be considered from the beginning. Standard and common-sense solutions are chosen over personal preferences. For instance: standard input/output files formats, APIs, models are used. Clear versioning and identification of third parties' version and their compatibilities are given. The optional nature of objects should be clearly identified.
- **"Think before you code":** check the literature, discuss with your colleagues, organize your information (sketches, lists, mind maps etc.). Optimise the computation time later, first use small examples. *Document-driven* development can be considered (write the documentation first, and then the code to make it true).