Programming guidelines at BME NTI Fusion Plasma Research Group

recorded by: Gergo Pokol (BME)
version 1.1 (24/2/2017)
based on: Robert C. Martin: Clean Code
following discussions at: 1/2/2017 Clean Code seminar (BME NTI R217)

Purpose

The aim is to have a **code that can be kept maintained for a long time by generations of developers with widely different skills**. Developers of the codes are typically students with gradually improving programming skills starting from an absolutely novice level. A realistic aim therefore is to provide a set of easy-to-follow rules mostly concerning the format and procedures, and a guideline to the programming principles that are to be followed by the more experienced members of the team.

Basic rules

Basic techniques of reproducible code development are to be followed. For each project we use an interlinked **version control system** (VCS) and an **issue tracking system**. Each commit must have an accompanying **commit message linking it to an issue** in the issue tracking system. On the other hand, each commit must be referenced in the corresponding issue. This also means that each development step (either bugfix or feature development) must **start by creating an issue**. Issue must be dealt with in **separate commits** to enable easy merge. One should always pursue a more-or-less **working master branch**. Major developments are to be done in separate **development branches**, while releases candidates are also to be treated in **release branches**.

The key to clean code is **continuous code cleaning and refactoring**, which is supported by **interactive development environments** (IDE) and a set of **comprehensive test suits**. For the sake of full compatibility and efficiency the **development environments are standardized** for each project: for Python we use *Anaconda+PyCharm*. Good coding practices and code coherence are facilitated by frequent **code reviews** and **joint programming sessions**. A general consideration regarding code quality is that less effort is made to improve code quality for components with limited scope and **more effort is put into cleaning up the core modules**.

Desired code format implies **lines not longer than 100 characters**, **functions as short as possible, but not longer than a screen** and source **file lengths not longer than necessary** (say a few screens). Files are to be organized into **modules based on functionality** and these modules are to be imported when necessary. Modules are to be **referenced by full name** to minimize the number of naming conventions to be learned. The code itself is to be **organized vertically using empty lines** following the recommendations of the IDE and maintaining coherence throughout the project. In general, formatting recommendations of the IDE are to be accepted, and even external tools might be used to comply with wide-spread conventions (e.g. *Pylint*).

Comments **duplicating information already present in the source code or in the VCS are strictly forbidden**, however, some comments may be necessary to **clarify the intention of the developer** and **make reference to the implemented formulas**.

As a general rule, the source **code should be readable** without any comments. The most important policy facilitating this is the application of the **naming policies**:

- **Variables** should be marked by **nouns** expressing the exact purpose of the variable. For **narrow scope one might use shorter names**, but for **broader scope variable names should really speak for themselves**.
- **Functions** should be named using **verbs**. For **narrow scope** (e.g. private functions) **short names** might do, but for **broader scope a longer name** might be needed to specify the exact functionality. Naming of functions is made easy by following the principle of **each function "doing only one thing"**.
- **Properties** should be named using **adjectives**. The name should **reflect the role of the property in the class** where it is defined, rather than its general purpose.

Naming policies depending on the scope implies that **whenever the scope changes re-naming might be needed**. **Naming problems might also suggest need for restructuring**: Very long names might indicate a need for a **class or module**. (Longish explanations might also be circumvented by employing **polymorphism**.) Similarly, a function needing many input and/or output variables of similar scope suggests the need for a **class or some compound data structure**. **Input variables should never be changed and returned**, always use separate variable(s) for output. Ideally, **any function should have just a few input variables**, that makes the code easy to read and the function easy to use.

An important tool to make the code flexible enough for long-term maintenance and development is the use of a **comprehensive test suit**:

- **Unit test suit** should **be developed for each function at latest when the general structure of the module has been consolidated**. Development of unit tests also help refactoring by indicating unnecessarily complex functions with multiple functionalities. Unit tests should be **automated** as much as it is supported by the IDE.
- **Integration tests** are to be developed whenever interaction of modules foreseen, especially in **case of using third-party modules**.
- For physics codes the **acceptance test** is mostly some kind of **benchmark to other existing codes** and the **conformance of the results to the general laws of physics and definitions of the physical quantities**. Development of acceptance tests is foreseen as a long-term project, but **vital before public releases.**

Data sets for testing, or any other purposes, must not be stored in the source code repositories, rather they must be **stored locally** and **downloaded on demand** from an online storage.

Error handling should be added where needed. As a general policy, it should always be implemented for **operations dependent on the run-time environment** that may result in errors specific to the deployment. Error handling should also be thoroughly worked out at the **input interfaces of modules to be released**.

User documentation of the code is only maintained for released versions and it is to be updated as part of the release process to ensure both timeliness and coherence. If deemed necessary, some kind of development documentation may also be prepared, but this should **avoid duplication of details of the code**.

Programming guidelines

In this section some general guidelines are pointed out. These should be attempted to be followed mainly in the process of consolidating the close-to-final structure the components. Some specific, yet still open development choices are also mentioned here.

In general, the structure should comply to the **SOLID principles** as detailed in Robert C. Martin: Clean Code. **Reuse of modules and functions** is encouraged, while also respecting the **independent deployability of components**.

In physics codes **units** are of central importance. A simple approach is to use **SI for all units** (with temperature measured in eV for plasma codes). Use of more advanced approaches, like the *units* module of the *Astropy* package (http://docs.astropy.org/en/stable/units/) in Python, might be considered. Similarly, the use of the *uncertainties* package (https://pythonhosted.org/uncertainties/) might be considered.

An easy was to improve **computational effectiveness** is to use **matrix operations** instead of for cycles. This implies the use of *numpy, scipy* and *numba* modules in Python.

Some **architecture schema** that can be considered is to be found at E. Gamma - Design Patterns: Elements of Reusable Object-Oriented Software.

Some more information on programming practices in Hungarian:
https://bme.videotorium.hu/hu/category/1840/szoftver-technologia
https://vik.wiki/Szoftvertechnol%C3%B3gia