# Good practice guide for code development in Artificial Intelligence solutions in health

Programa Salut/IA

TIC Salut Social

Generalitat de Catalunya



This report has been elaborated by the Artificial Intelligence Area of the TIC Salut Social Foundation

Authors: Susanna Aussó, Didier Domínguez and Yeray Bartolomé.

Electronic edition: December 2022

# /Table of contents

# 1. Introduction

The Programme for the Promotion and Development of Artificial Intelligence in the Catalan Health System aims to create an environment to aid the development and implementation of Artificial Intelligence (AI) solutions for the optimization of processes in the Catalan health system.

TIC Salut Social Foundation has created this guide to support those involved in the development of code for Artificial Intelligence algorithms applied to the field of health.

The follow-up of this set of good practices makes it possible to obtain a code that is more understandable, reusable, scalable and easier to modify that will help reduce the time spent on review tasks and prevent common errors during the different stages of the project. These recommendations also aid collaborative work, as they allow other collaborators who also develop code to quickly understand and, if necessary, develop new advances in the project following a specific methodology.

Coding standards are a critical part of maintaining good practices during software development. We understand a coding standard as a set of established rules and recommendations (names, formats, etc.) for writing code that vary depending on the programming language chosen. These standards have been emerging from various developer communities over time, and if properly applied, they can significantly increase code quality and management.

Artificial Intelligence (AI) covers a wide range of techniques, methods and approaches that can be implemented in multiple programming languages. Due to the increasing complexity of AI applications in health care and their rapid expansion in recent times, the adoption of consistent coding standards in software development has become necessary.

There is currently no general set of requirements that software for medical devices must meet, either from international standardisation organisations or from key medical regulators such as the European Medicines Agency and the Food and Drug Administration in the USA. Regulators require device developers to adequately validate the security and cybersecurity of their technologies, but details on how device security should be achieved are scarce. Therefore, while there are no clear guidelines that directly mandate the use of certain coding standards, it is clear that as code bases become more complex and connected, coding standards will become more relevant to programming teams.

# 2.

# AI programming languages

There are many programming languages capable of meeting different needs in the design and development of different software intended for AI. These may be data storage and querying, data analysis, model training or explainability techniques. These languages often include a range of these functionalities simultaneously.

The TIOBE index [1] is a programming language popularity indicator, which this company has created with the participation of the programming community. The index is updated once a month. The ratings are based on the number of skilled engineers worldwide, courses and third-party vendors.

| Oct 2022 | Oct 2021 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 1 | | Python | 17.08% | +5.81% |
| 2 | 2 | | C | 15.21% | +4.05% |
| 3 | 3 | | Java | 12.84% | +2.38% |
| 4 | 4 | | C++ | 9.92% | +2.42% |
| 5 | 5 | | C# | 4.42% | -0.84% |
| 6 | 6 | | Visual Basic | 3.95% | -1.29% |
| 7 | 7 | | JavaScript | 2.74% | +0.55% |
| 8 | 10 | ▲ | Assembly language | 2.39% | +0.33% |
| 9 | 9 | | PHP | 2.04% | -0.06% |
| 10 | 8 | ▼ | SQL | 1.78% | -0.39% |
| 11 | 12 | ▲ | Go | 1.27% | -0.01% |
| 12 | 14 | ▲ | R | 1.22% | +0.03% |

*Figure 1. TIOBE INDEX*

In order to simplify the task of choosing the languages to use, this guide focuses on Python, Java, C++ and R, since they are among the most-used medium- and high-level languages in the software industry [Fig. 1], and R is one of the languages dedicated to data processing and visualisation most used in scientific research. A large amount of documentation and projects are available for all of these to aid in the task of selecting the most appropriate for the software's needs. Nevertheless, it is worth knowing that there are other languages used rather often in AI such as Julia, Haskell, Lisp, JavaScript, Prolog and Scala. These are more modern languages with a significantly lower amount of documentation in this field, although it is advisable to consider them.

## 2.1 Python

Python is a fourth-generation, high-level and interpreted programming language [2]. This means that it does not need to be compiled beforehand but can be run directly with an interpreter. This has the advantage of being more independent of the hardware and execution environment, and it is possible to execute the code remotely in real time, as in the example of Google Colab. With regard to variables, it is an untyped language with discrete variable declaration, as you only need to assign a value to a previously non-existent variable to initialise it. The interpreter automatically assigns the type at runtime based on the assigned values. In terms of code hierarchy and nesting, this language does not use brackets (unlike Java, C++, JS, C#, etc.) but instead indentation (tabs).

Python lets you load and process data, visualise them, generate basic and complex statistics, and develop neural networks, genetic algorithms, etc. Aside from its popularity due to its simplicity and versatility, the main reason to use Python to develop AI is the ecosystem of resources available. For example, the most well-known and widely-used deep learning and machine learning packages and tools such as Pandas, NumPy, TensorFlow, Keras, Scikit-learn and many more are all developed in Python.

The language has evolved in parallel with the discipline of AI, and this makes it almost indispensable to know it if you work in this field. In addition, it provides very clear code, allows rapid prototyping, and aids collaboration for teamwork.

## 2.2 R

Like Python, R is a fourth-generation, high-level, interpreted programming language [3]. They also share discrete variable declaration and typing. While it has traditionally been associated with more academic environments, it is a specific language for working in data analysis and machine learning.

It is an open-source, cross-platform, functional, object-oriented and easy-to-learn language. It has a very popular development environment for the language called RStudio (which now also supports Python).

## 2.3 Java

Java[1] is a third-generation language with a high level of mixed interpretation, because it is precompiled in binary but executed by an interpreter called the Java Virtual Machine (JVM) [4]. Java is a strongly statically-typed language with explicit variable declaration. Intended for creating scripts, macros and similar material, its main advantage is its ease of learning for common tasks and that it runs both in browsers and on servers (via platforms like Node.js).

It is an excellent complement to other programming languages such as Python and R, especially for visualisation. It is a multi-paradigm language that follows the principles of being object-oriented and Write Once/Run Anywhere (WORA). This principle means that Java is a language that can be run on any platform that supports it without the need for compiling. It is suitable not only for search algorithms or natural language processing but also for neural networks. This is possible thanks to the JVM (Java Virtual Machine), a kind of emulator that simulates a common architecture within each OS for which it is distributed. This allows decoupling of the OS and the application to be executed.

## 2.4 C++

C++ is a precompiled third-generation language [4]. It is characterised by being medium- and high-level, because it usually runs on the operating system, but it offers aspects ranging from low-level functions inherited from C, to object-oriented programming and web programming. C++ is the fastest of the programming languages mentioned, and its speed is valued for time-sensitive AI programming projects. It provides fast execution and a low response time, which is applicable to search engines and computer game development. This is because C++ is an evolution of and written in C. It is therefore designed to provide low-level implementations that make the most of the hardware available in exchange for greater coupling to the architecture supporting the code. In addition, C++ allows extensive use of algorithms and is efficient in the use of statistical AI techniques. Another important factor is that C++ supports the reuse of programs in development due to inheritance and data hiding. It is therefore time-efficient and cost-saving.

Its extensive libraries are ideal for complex AI code, classification, faster mathematical calculations, and high-performance applications.

[1] https://www.java.com/es/

## 2.5

# Performance and ecological footprint of programming languages

The table below summarises the consumption in Watts (w) of the languages mentioned, based on different amounts of inputs per measurement:

| ALGORITHM | PYTHON | R | C++ | JAVA |
|---|---|---|---|---|
| Heap Sort (100 inputs) | 5.4 | 6 | 17.3 | 5.2 |
| Heap Sort (1000 inputs) | 8.1 | 9.4 | 17.6 | 8.2 |
| Heap Sort (1500 inputs) | 13.4 | 12.8 | 18 | 14.1 |
| Quick Sort (100 inputs) | 6.8 | 8.1 | 17.2 | 6.2 |
| Quick Sort (1000 inputs) | 14.4 | 15.3 | 17.8 | 14.2 |
| Quick Sort (1500 inputs) | 16.3 | 16.6 | 20.2 | 17 |
| Selection Sort (100 inputs) | 6.1 | 6.4 | 17.4 | 5.8 |
| Selection Sort(1000 inputs) | 6.7 | 7.1 | 18 | 6.9 |
| Selection Sort(1500 inputs) | 8 | 8.2 | 20.2 | 8.3 |
| Matrix Addition (2x2) | 0.8 | 1.1 | 17.7 | 0.7 |
| Matrix Addition (5x5) | 2.2 | 2.3 | 18.9 | 2.4 |
| Matrix Addition (10x10) | 3.1 | 3.2 | 19.6 | 3.2 |
| Matrix Multiplication (2x2) | 0.9 | 1 | 17.5 | 0.7 |
| Matrix Multiplication (5x5) | 1.6 | 1.8 | 18.2 | 1.6 |
| Matrix Multiplication (10x10) | 2.8 | 3 | 18.6 | 2.9 |

*Table 1. Energy consumption of different algorithms in the languages mentioned.*

As one can see, out of the languages mentioned, C++ is the one with the highest base energy consumption. However, it is also the one that grows the least in proportion to the volume of inputs provided. Meanwhile, Java, Python and R, when interpreted, consume a much smaller volume of Watts when it comes to simple operations. However, this scales linearly with the number of inputs. In any case, Python is considered the fastest and most energy-efficient language for most coding tasks.

# 3. Software development methodologies

Today, most software projects must support dynamic and iterative development, while being a feasible and profitable tool for a changing, scalable and migratable project. For these reasons, in the industry itself, the work of notable people such as Hirotaka Takeuchi and Ikujiro Nonaka [5] and Jeff Sutherland [6, 7] and Ken Schwaber [8] has modelled so-called Agile software development philosophies. The Agile paradigm, which focuses on short development cycles in which all of the activities are performed simultaneously (discovery of requirements, implementation, testing, including the review of the work in previous cycles), is a philosophy that was "proposed to overcome the convolutional development methods' limitations" [9]. In other words, it is aimed at minimising the impact of cascading development, allowing some flexibility when making changes to requirements in each development stage. This way of working is a good practice that minimises subsequent errors in the software in production.

Within the Agile philosophy, over the years, several development methodologies have been elaborated and established, each with its own advantages and weaknesses, with the aim of adapting this philosophy to the nature of different software projects, development teams, and organisational policies. Three of these stand out:

- **Scrum methodology:** focused on being a framework that divides project development into several timeboxed sprints. Within each sprint, different tasks related to the development of the planned functionalities must be carried out, such as reviewing upcoming requirements, implementing present requirements, and integration and/or refactoring of previous ones, comprehensive product testing before releasing the new version, and writing documentation, but emphasising the code as the main source of it.
- **Kanban methodology:** this consists of showing the tasks in each iteration on labels that can be moved in columns according to their state of development. It focuses on showing the progress of the product to people outside the development team.

- **Lean Methodology:**
  - Eliminating waste/leftovers: By waste or leftovers we mean everything that does not add value to the software. Within software development we could include the following elements: generated code that offers unwanted or unneeded functionality, delays in the software development process, problems with internal communication, excessive documentation, etc. However, in AI topics we sometimes work with non-deterministic or highly-complex libraries and algorithms and we have to take into account a couple of concepts and best practices. When debugging is difficult, it is often a sign that the code lacks diagnostic functionality or traceability. This aspect of writing code is often overlooked because it is not functional. However, it can save hours and sometimes even days of debugging time.
  - Strengthening the team: Helping developers participate in making decisions about the time associated with tasks, prioritising them and other ways to involve team members, making it easier for them to feel an important part of it. In addition, the developers themselves know first-hand which tasks cost more, which cost less and the implications they have for the project's lifecycle. Techniques in which weights, efforts or complexities are assigned to the tasks to be performed come within this principle.
  - Continuous Integration: Having a good continuous integration system that includes automated testing, builds, and usability testing is critical because it makes it possible to produce software that is easy to maintain, improve and reuse. This avoids adding leftovers to the software and aids maintenance of the code and subsequent versions in production.
  - Viewing the whole: Analysing our software's interactions with the rest of the systems allows us to study possible improvements and changes that result in better performance and bring greater value to end users and the project team.


# 4. Recommendations for code development

We define five main concepts in the application of good programming practices: nomenclature, order, profitability, style and documentation. Each of them includes different sections:

- **Nomenclature**
  - Naming conventions for variables
  - Naming conventions for classes and functions
- **Order**
  - Add clear and concise comments to the code
  - Grouping and organisation of code
- **Style**
  - Limit line length
  - Avoid using “magic numbers”
  - Indentation
  - Avoid deep nesting
- **Profitability**
  - Portability
  - Reusability and scalability
  - Test to verify functionality
- **Documentation**
  - Documentation and/or README file with explanation and guide used by the code

These concepts, which make up the general structure of good practices, are explained in detail below.

# 4.1. Nomenclature

### Naming conventions for variables

During programming, variable names must be easy to understand and represent the data they store. Therefore, the way variables are named is key to making code readable and avoiding confusion.

The idea behind variable naming during code development is simple: create variable names that are self-explanatory and consistent throughout the code. Trying to save time by using very short names for variables and functions is counterproductive in the long run, especially when there are many code editors and IDEs[2] available to help write code.

Names must have word limits. There are three popular options:

- **UpperCamelCase or PascalCase:** The first letter of each word is capitalised (e.g. ParseRawImageData()).
- **camelCase:** The first letter of each word is capitalised, except the first word (e.gparseRawImageData()).
- **Underscores:** underscores between words, such as mysql_real_escape_string().

Some platforms tend to use a particular naming scheme. In the case of Java and C++, the most common convention is to use camelCase for variables and methods, while PascalCase is used for constructors and class names. In the case of Python and R, underscores are used for variable names.

These styles can also be mixed. Some programmers prefer to use underscores for procedure functions and class names, but they use CamelCase for class method names.

[2] An Integrated Development Environment, unlike an editor, is a heavier program that requires much more RAM and a more powerful processor.

### Naming conventions for classes and functions

The basic concepts of naming classes and functions are an essential aspect of learning to program.

Similar to variable naming conventions, functions and classes should also consist of descriptive titles that are delimited using conventions, as mentioned above. The purpose of using suitable naming conventions is to ensure that variables, functions, and classes in the code can be easily distinguished from one another.

## 4.2. Order

### Add clear and concise comments to the code

Code always ends up being modified or updated over time. Almost every programmer will come across someone else's code at one point or another. A bad habit among inexperienced programmers is to include few comments during software development. This poses a significant challenge when working in a team, where more than one person may be working on a particular module.

On the other hand, it is also advisable not to overdo it. Too many comments can decrease the value of knowledge transfer between developers working on the same module. Ideally, comments should explain why certain code is being used. If a comment of more than one line must be written to explain what the code is doing, you should consider rewriting the code to make it more readable.

In short, there is a great debate about whether it is a good standard to code comment or not. Nowadays, a significant part of the technology industry considers a good code to be the one that by using proper nomenclature, structure and formatting, is readable without the need for comments. In any case, it is worth explaining that comments are not the right way to convey information about the functionality and structure of the project, and that they can in no way replace clean code and good documentation.

### Grouping and organisation of code

Good organisation is vital to maintain a good structure and make the code understandable. Very often, certain tasks require several lines of code. Adding a comment at the beginning of each block of code emphasises visual separation and aids understanding. Keeping similar tasks inside separate blocks of code, with gaps between them, as shown in the following example, is very positive for good code management:

```
# llibreries
import argparse
import os
import sys
import numpy as np
import pandas as pd

# moduls
# carregar llistat de gens
def load_genes(genes):
        list_genes = open(str(genes), "r")
        genes = [ ]
        for line in list_genes:
                line = line.strip()
                genes.append(str(line))
        return genes
```

*Figure 2. Example of code grouping and organisation.*

## 4.3.

# Style

A programming style is a set of guidelines used to format programming instructions. Following a style makes code easier for programmers to understand and maintain, and helps reduce the likelihood of introducing errors. Guidelines can be developed from the coding conventions used in an organisation with style variations for different programming languages.

Key elements of the programming style guide include naming conventions, the use of comments, and formatting (indentation, whitespace, etc.). In some languages (e.g. Python) indentation is used to indicate control structures (so proper indentation is required), while in other languages indentation is used to improve visual appearance and readability of the code (e.g. Java).

### Limit line length

It is good practice is to avoid writing lines of code that are too long horizontally, as our eyes feel more comfortable reading tall and narrow columns of text. For example, when using the Python language, the recommended line length limit is 79 characters.

```
#unload alternatives
refbases = set(refBase.split (‘,’))
altBases = set(altBase. split (‘,’))
added = set ()
for refBase in refBases.difference(altBases):
        for altBase in filter (lambda x: x != refBase, altBases):
                outline = process_alt(refBase, altBase, contig, pos,

                out.write(outline)
                added.add((refBase, altbase))
for altBase in altBases.difference(refBases):
        for refBase in filter(lambda x: x != altBase, refBases) :
                if (refBase, altBase) in added:
                        continue
                outline = process_alt(refBase, altase, contig,
```

*Figure 3. Example of line length limitation.*

## Avoid using “magic numbers”

The concept of “magic numbers” in programming refers to the use of numerical values encoded in the code. Using these numbers may make sense to the person writing the code, but it can make it difficult to understand the purpose of that number when the programmer looks at the same piece of code in the future, or when someone else does.

```
#cas 1
thr = 400
for gene in results:
  p = results[gene]
    if p <= thr:
      print gene, p

#cas 2
for gene in results:
  p = results[gene]
    if o <= 400:
      print gene, p
```

*Figure 4. Example of how not to use magic numbers.*

Looking at the code in Figure 4, in the first case it is clear that it is checking if the count is less than the selected threshold (thr = 400), while in the second case the meaning of the number 400 is unknown. In addition, it is easier to update the threshold thr by changing its value in a single place, which would not be possible using magic numbers, because you would have to go through the entire code to understand where the value comes from.

## Indentation

Formatting and indentation are necessary to organise code. By using indentation, whitespace and tabs within code, programmers ensure that their code is readable and organised.

There is no right or wrong way to indent code. There are popular opinions but no pattern is universally followed. The important thing is to be consistent with the chosen style. Changing indentation styles in the middle of a script leads to confusion and errors in code execution. In some languages (e.g. Python) a combination of different indentation styles (tabs and spaces) is not allowed. If you are part of a team or contributing code to a project, you must follow the existing style that is being used in that project.

Indentation styles are not always completely different from each other. Sometimes they mix up different rules. For example, in the R language, the opening curly brace “{“ in a function goes on the same line as the control structures, but after the functions have been defined, they go on the next line and in an more leftward position, as shown in the following image:

```
y <- 10
y <- function(x) {
   x+y
}
f (5)
```

*Figure 5. Indentation and use of brackets.*

### Avoid deep nesting

Too many levels of nesting can make code hard to read and follow. To improve readability, it is usually possible to make changes to the code to reduce the level of nesting:

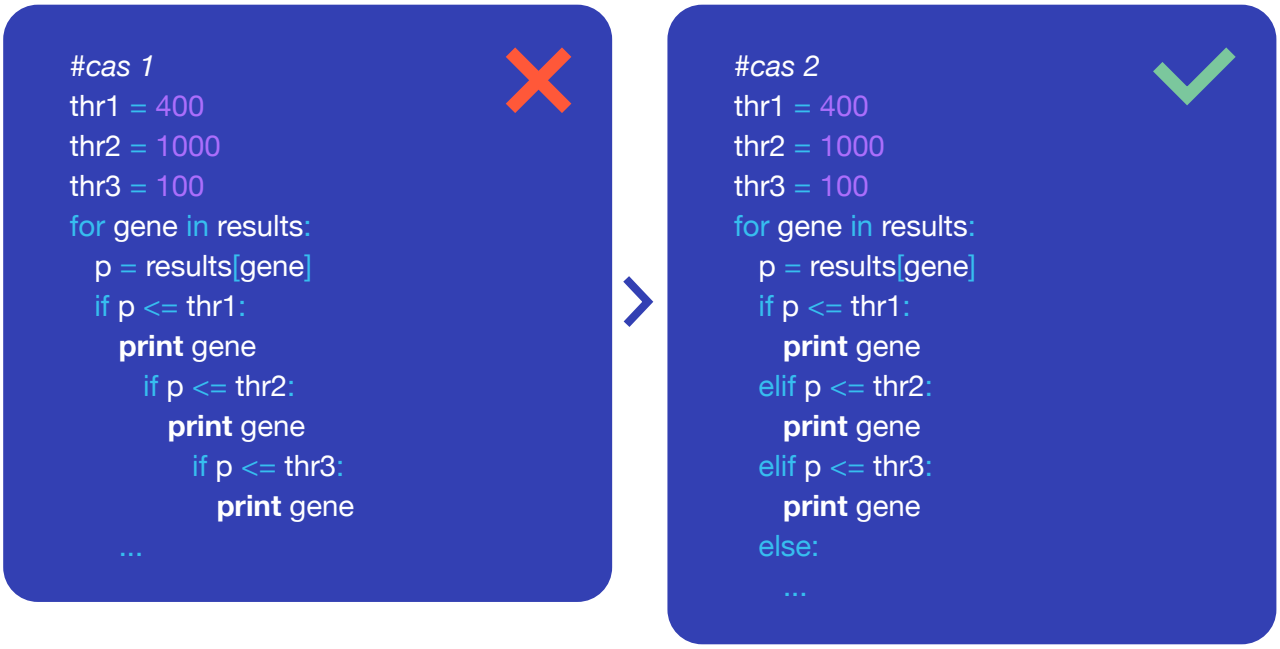


*Figure 6. Example of best practice in code nesting.*

## 4.4.
## Profitability

### Profitability

Portability is the ability of the source code to run on different machines and platforms as much as possible. If programmers have to rewrite the same code when software is transferred from one environment to another, time and effort are wasted. Multi-platform support can be planned early in software development; you can write code that could work in every possible environment.

Portability is a key aspect that ensures the functionality of a program. If the code contains specific values for environmental parameters in the source code (hard-code), such as usernames, hostnames, IP addresses and URLs, it will not run on a host with a different configuration. In these situations, the developer would have to change the source code and that would not help portability. To deal with this, variables would need to be “parameterised” and set before running the software in different environments. This can be controlled with properties or settings files, databases or application servers.

Also, resources such as XML files must also have variables instead of literal values. Otherwise, references would need to be changed while coding each time you want to port your application to another environment.

Another good practice is the creation of portable and self-sufficient application containers with Docker. Containers allow you to build an application with all the libraries and dependencies it needs and distribute it as a single package. The application will run on any other machine in the production environment, even if the machine's settings are different from the machine used to write the code.

### Reusability and scalability

In code development, reusability is an essential goal of software design. If modules and components have already been tested, you can save time by reusing them. Software projects often start with an existing framework or structure that contains a previous version of the project. So development time, costs and resources can be reduced by reusing software components and modules. This translates directly into faster delivery of the project, thus increasing profitability.

Another key aspect to pay attention to is code scalability. As user demands change, new features and improvements are constantly added to an application. Therefore, the ability to incorporate updates is an essential part of the software design process. Accordingly, it is good practice to test the code of AI algorithms with small subsets of data. Once you are certain that there are no errors, you can scale up with data of a larger size.

### Test to verify functionality

Testing the work while coding is a vital part of software development and should be well planned as it requires test cases to be ready before software development begins. Also, while basic unit testing is a good practice to adopt, it is also useful to perform automated functional testing with the help of tools such as Unittest, Doctest, Geb, Spock and Selenium.

Once the software is finished and made available to the end user, depending on its purpose, it is recommended to provide a test executable for when the user installs the software on their local environment This makes it easy to test whether or not the user's environment is suitable before they start performing more complex tasks.

## 4.5. Documentation

Good documentation of everything that has been carried out is essential for the correct execution of the developed software. The basic documents in order to have a well-ordered, executable and easily-accessible project for future versions are a README, INSTALL and LICENCE. These documents contain the structure of the code, the guides that have been used to program it, the steps to follow for correct installation, and the terms of the license whereby the software is made available to the user.

# 5.
# Coding standards

A coding standard is a programming paradigm that seeks to reduce the number of decisions the programmer must make when writing code. Companies such as Google, whose primary programming language is Python, may create their own coding standards to ensure consistency in naming conventions and code layout. Also, free software libraries (such as Tensorflow[3]) may use previously-established standards and adapt them to their needs. These guides often end up being a reference for programmers and must be taken into account.

The following are the most widely recognised and used standards for programming in R, Python, C, C++, and others, which make code easier to read, share and verify.

## 5.1. Programming standards in R

### Tidyverse Style Guide

The Tidyverse Style Guide[4] by Hadley Wickham consists of a set of rules and recommendations for programming in R safely and correctly. It is based on two main sections: analysis and creation of libraries. The analysis section focuses on nomenclature, syntax, the creation and structure of functions, and the correct operation of pipes (the process of going from one function to another). With regard to building and distributing libraries, it also focuses largely on naming and organising files, good documentation and description of functions, creating a test file for code validation, and the correct communication of errors in the code functionality (see Annex 1).

### Google R Style Guide

Google's R Style Guide[6] is a fork of the Tidyverse Style Guide by Hadley Wickham, mentioned above. Google's modifications were developed in collaboration with the internal R user community and reaffirm Hadley Wickham's rules and recommendations for efficient and correct R programming. The two main differences are the identification of functions with BigCamelCase to clearly distinguish them from other objects, and the explicit use of the return() option, among others (see Annex 2).

### R Coding Style (Amazon)

Amazon's R Coding Style R Coding Style is based on the Tidyverse Style Guide, Google's R Style Guide (both mentioned above), and Hadley Wickham's "Advanced R" guide [10]. Due to the generation of vast amounts of content on its platform, Amazon decided

[3] https://www.tensorflow.org/community/contribute/code_style
[4] https://style.tidyverse.org/index.html
[5] https://google.github.io/styleguide/Rguide.html
[6] https://rstudio-pubs-static.s3.amazonaws.com/390511_286f47c578694d3dbd35b6a71f3af4d6.html

to create its own guide that includes a series of recommendations on language, style and good use and which is constantly updated. This guide contains aspects of nomenclature, syntax and organisation (see Annex 3).

## 5.2. Programming standards in Python

### Python Enhancement Proposals: PEP8

The main benefit of programming in Python is that the language is easy to read. The PEP8 guide, written by Guido van Rossum, Barry Warsaw, and Nick Coghlan, provides a set of rules and recommendations for Python programming to keep it easy to read and write. This set of recommendations has evolved and been updated over time, and has been adapted to the evolution of the Python language. The guide contains concepts ranging from syntax to the structural organisation of the code (see Annex 4).

### Google Python Style Guide

Google's Python Style Guide[8] contains all the rules and recommendations for programming in Python. Each open source project has its own code guide and best practices. They have produced their own guide containing a series of recommendations on language, style and good use (see Annex 5).

### The Hitchhiker's Guide to Python

The Hitchhiker's Guide to Python[9] is an unofficial public guide that is constantly updated by Python programmers, which provides you with a best practice manual with recommendations for correct installation, configuration and use of Python [11].

## 5.3. Programming standards in C,C++ and others

### MISRA C/C++

MISRA[10] provides world-leading best practice guidelines to provide source code portability, safety and security in the context of embedded software. It began in the early 1990s as a project part of the UK government's SafeIT programme, which developed guidelines for the creation of software embedded in road vehicle electronic systems. Over time, MISRA has continued to work on a voluntary basis, producing reference publications such as MISRA C and MISRA C++ [12, 13].

Currently, MISRA C:2012 is used. This was published in 2013 and updated in 2020 and consists of 17 directives, which include general compliances not related to the source code (requirements, specifications, design, etc.), and 158 rules, which include information related to the source code (see Annex 6).

### CERT

A Computer Emergency Response Team (CERT[11]) is a group of computer security experts responsible for protecting, detecting and responding to an organisation's cybersecurity incidents.

CERT/CC coding standards consist of rules and recommendations that enable developers to avoid unsafe coding practices and undefined behaviours that can lead to

[7] https://peps.python.org/pep-0008/
[8] https://google.github.io/styleguide/pyguide.html
[9] https://docs.python-guide.org/
[10] https://www.misra.org.uk/
[11] https://wiki.sei.cmu.edu/confluence/

vulnerabilities. These are under constant review, with the latest rules and recommendations available online[12].

The organisation has developed four subsets of rules and recommendations to support four programming languages: CERT C, which focuses on helping reduce the likelihood of C language vulnerabilities; CERT C++, which shares many rules and recommendations with CERT C but adds specific ones related to C++ in order to avoid vulnerabilities specific to it; CERT Oracle for Java, which consists of specific rules and recommendations for Java to eliminate possible vulnerabilities in robust systems; and CERT Perl, which focuses on standards and recommendations to avoid Perl language vulnerabilities.

### CWE

CWE[13] (Common Weakness Enumeration) is a community-developed list of software weakness types related to security problems. These "weaknesses" are flaws or errors in software implementation, code, design or architecture which, if not resolved, could make systems, networks or hardware vulnerable to attack.

CWE's primary goal is to stop vulnerabilities at source by educating software architects, designers, programmers and buyers on how to eliminate the most common errors before delivering products. Using CWE helps prevent the types of security vulnerabilities that can affect the software and hardware industries, evaluate coverage of tools targeting these weaknesses, leverage a common baseline standard for weakness identification, mitigation and prevention, and prevent software and hardware vulnerabilities prior to deployment.

## 5.4.

## Benefits of coding standards in software development

In a team of programmers without established code enforcement, it is very common for them to program individually without following unified criteria. This results in a repository that is hard to understand and maintain. Poorly-organised code can result in an increase in working hours to search for the source of an error, and create divergences between the work team when the work is unified.

Implementing coding standards in software development brings benefits such as cultivating a culture of excellence, increasing team effectiveness and efficiency, minimising errors, rework and delays, and creating a positive feedback loop, as detailed in the following points:

### They cultivate a culture of excellence

ECoding standards cultivate a culture of excellence in which the goals and outcomes are clear to all of the team members. They are independent, documented, widely understood and achievable in a clear and demonstrable way. It no longer depends on one person's style or understanding of what is ideal or what the current team culture is.

Over time, software development practices become more efficient as the team learns to code the standard in a unified and systematic way.

### They increase effectiveness and efficiency

Industry-accepted standards not only identify problems to avoid, but also provide best practices to help increase teams' efficiency.

[12] https://wiki.sei.cmu.edu/confluence/dashboard.action#all-updates
[13] https://cwe.mitre.org/index.html

Apart from identifying security vulnerabilities and possible flaws to avoid, they advise on best practices and styles. All of the rules and recommendations in coding standards continue to be reviewed, updated and expanded to cover new issues and scenarios (see section 6.2).

Software development teams can benefit from prior industry experiences and the work of thousands of experts to refine their practices and produce code with the highest possible security and trustworthiness. Following these best practices makes it possible to produce more efficiently and quickly, and avoid problems.

### Minimising errors, rework and delays

When you are about to launch a release, a flaw or security vulnerability is often discovered that inevitably requires a return to development and a restart of the process to fix it before release. Alternatively, there are also risks that software containing a defect will be delivered to the customer which, once identified and analysed, will require the team to abandon their current tasks to fix and update the product. The consequences of such a situation can be even more serious in the case of critical industries, as they may be very visible or cause serious security problems.

In short, full compliance with coding standards avoids costly common mistakes throughout the development process, while avoiding rework and last-minute delays.

### Positive feedback loop

Compliance with standards creates a positive feedback loop of fewer defects in the field, an increased competitive advantage through adherence to standards, and decreased time to market for new products.

Teams want to work on new innovative features for dynamic and expanding products. Committing to a recognised coding standard is a competitive advantage, retaining current users and opening doors to new ones. As software problems and security vulnerabilities become more visible, instability and user insecurity increases, and they are intended to reduce risks.

When more reliable and industry-compliant code is delivered, with fewer defects in the field, the team can deliver new features and new products more quickly. At the same time, this improves the brand's reputation and creates future opportunities. And so the feedback loop begins.

For many years, the medical device industry was motivated to seek coding standards for this very reason, largely based on the MISRA C standards (see Section 6.2.3). Over time, industry-specific standards have emerged such as the FDA's recognition of UL2900 for security in medical devices and IoT networks.

Choosing the right coding standard is important because:

- Code trustworthiness is increased by enforcing compliance with rules.
- It educates developers on secure code development practices, reducing time and the costs of onboarding and training.
- It establishes a consistent approach to code analysis that can be shared across teams.
- It provides flexibility to adapt to different needs, without building new code development rules from scratch.

## 5.5.
## SOLID principles

Today, in most high-level software projects, due to the high level of abstraction, object-oriented programming (OOP) is used at least in part. In order to create a standard of good coding practices that would help produce quality object-oriented code, the SOLID principles were introduced in the early 2000s. This mnemonic reminds you of the five principles that make a solid object-oriented program. These principles are:

- **Single responsibility:** An object should only have a single reason to change.
- **Open/closed:** Software units must be open to extension and closed to modification.
- **Liskov substitution:** A program's objects should be replaceable by instances of their subtypes without altering the program's correct operation.
- **Interface segregation:** Many client-specific interfaces are better than a general purpose one.
- **Dependency investment:** The notion that you should rely on abstractions, not implementations. Dependency injection is used to follow this principle..

# 6. Code quality

Quality assurance is a systematic way of creating an environment to ensure that the software being developed meets quality requirements. Therefore, it is a preventive process aimed at establishing the correct methodology and standard to provide a quality environment for the product being developed.

There is no single way to measure code quality. What is valued may vary between development teams, but some key traits to measure for higher code quality are [14]:

- **Functionality:** How effectively the software interacts with other components of the system. The software must provide appropriate functions as per requirement, and these functions must be implemented correctly.
- **Reliability:** Reliability is the capability of the software to perform under specific conditions for a specified duration.
- **Usability:** Usability of software is defined as its ease of use, i.e. how easily a user can understand the functions of the software and how much efforts are required to follow the features.
- **Efficiency:** The efficiency of the software is dependent on the architecture and coding practice followed during development.
- **Maintenance:** Maintainability measures the ease with which the software can be maintained. It is related to the size, consistency, structure and complexity of the codebase.
- **Portability:** Portability measures how usable the software is in different environments. This is a matter of platform independence, i.e. how easily a system adapts to changes in specifications, how easy it is to install the software and how easy it is to replace a component in a given environment.

## 6.1. Static code quality analysis

Developing quality code takes time and effort in the early stages, but in the long run it reduces the cost of maintenance and bug fixes. Analysing and measuring code quality can be tricky, as it can be subjective. However, certain metrics can be used to objectively assess it, and there are several ways to reduce complexity and improve quality.

Static code analysis identifies defects, vulnerabilities, and compliance issues as you code. It detects problems that are often overlooked when using other methods such as compilers and manual code reviews. With static code analysis, software problems can be fixed earlier, reducing overheads and enabling a quality product to be delivered on time. It consists of a series of automated checks that are performed on the source code. A static analysis tool scans the code for known common errors and vulnerabilities, such as memory leaks and buffer overflows[14]. The analysis can also enforce code development rules.

Like all forms of automated testing, static code analysis ensures that checks are performed consistently and provides quick feedback on the latest changes. However, it can only identify cases in which programmed rules are broken; you cannot find all the bugs just by re-

[14] A buffer overflow or buffer overrun occurs when more data is placed in a fixed-length buffer than it can handle. The additional information, which has to go somewhere, may overflow into adjacent memory space, corrupting or overwriting the data contained in this space.

ading the source code. There is also the risk of false positives, so it is necessary to interpret the results.

So static code analysis is a valuable complement to code reviews, as it highlights known issues and frees up time for tasks such as design review and the general approach.

The **benefits** of using automated static analysis include:

- Processing tens of thousands of files in a few minutes. A human reviewer is typically only able to review a few hundred lines of source code per hour, which dramatically shortens the overall project lifecycle and increases the burden on work teams.
- It is less expensive than manual code inspections. It identifies known defects on demand and never misses anything. Manual code inspections are less effective, more time-consuming, and more expensive.
- They provide more accurate results than a human. They eliminate errors and omissions that occur during manual code reviews, thereby improving code quality.

Regarding the **drawbacks**, organisations should be aware of the following:

- False positives can be detected.
- A tool may not indicate what the flaw in the code is when it finds one.
- Not all code development rules can always be followed, for example rules that require external documentation.
- It cannot detect how a function will be executed.
- It may not be possible to parse system and third-party libraries.

## 6.2.
## Code quality metrics

Cyclomatic complexity and Halstead complexity are two quantitative static analysis metrics to measure code quality.

### 6.2.1.
### Cyclomatic complexity

Cyclomatic complexity (CC) is a metric that quantifies software quality. It was developed by Thomas J. McCade in 1976 and uses a simple calculation based on a control-flow diagram representing the code to be analysed [15]. This flow diagram shows the number of linearly independent paths that the code takes when it is executed and is represented by nodes and edges. Each node represents a basic block of code and the edges represent the connections or paths between blocks of program code (Fig. 7).

A

```
int x,y,r;
if (x<0 || v<0) {
   system.out.println('X o Y son negatius');
} else {
   r=(x+y)/2;
   system.out.println('La mitjana de X i Y es:' + r ):
}
```

B

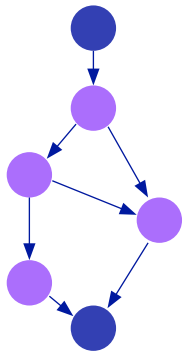

*Figure 7. Example of a Cyclomatic Complexity calculation. A: fragment of code to be analysed; B: diagram of representative nodes and edges when executing the code fragment in Figure 7A. The blue nodes represent the input and output of the execution, the orange ones the conditions, and the edges the execution paths.*

Complex code is unreliable, inefficient and of low quality, so it is important to measure cyclomatic complexity so you can improve code quality if necessary. The higher the number of paths, the more complex the code, and the more likely it is to contain defects and be difficult to test, read, and maintain.

Taking the code fragment in Figure 7A as an example, the calculation to measure cyclomatic complexity (CC) is:

$$CC = no.\ of\ edges - no.\ of\ nodes + 2*no.\ of\ output\ nodes$$

As one can see in Figure 13B, the blue nodes (2) represent the input and output of the execution, the orange ones (4) the conditions, and the edges (7) the execution paths. Therefore, the resulting formula would be:

$$CC = 7 - 6 + 2 = 3$$

There is agreement regarding the simplicity of code that depends on the value obtained when calculating its cyclomatic complexity. Based on the value obtained, we can determine if the code is simple (CC value between 1-10), slightly complex (CC value between 11-20), complex (CC value between 21-50), or not testable (CC value of more than 50). When the code has a value between 1 and 10 it is considered clean, testable, effective and manageable. In contrast, above 10, the risk of defects gradually increases, until it reaches 50, which is considered untestable.

## 6.2.2. Halstead Complexity

Halstead's complexity measure was developed by Maurice Halstead in 1977 as a means to determine a quantitative measurement of complexity from operators and operands using indicators [16]. This metric measures the computational complexity of the code in terms of bugs, difficulty and testing time.

Halstead metrics are only meaningful at source code level, and vary with the following parameters:

| PARAMETER | MEANING |
|---|---|
| $n_1$ | Number of distinct operators |
| $n_2$ | Number of distinct operands |
| $N_1$ | Number of occurrences of the operator |
| $N_2$ | Number of occurrences of operands |

*Table 2. Parameters and meanings of Halstead metrics.*

Taking into account these parameters, various metrics can be measured:

| METRIC | MEANING | FORMULA |
|---|---|---|
| n | Vocabulary | $n_1+n_2$ |
| N | Length | $N_1+N_2$ |
| V | Volume | $N*\log_2 n$ |
| D | Difficulty | $(n_1/2)*(N_2/n_2)$ |
| E | Effort | V*D |
| B | Bugs | V/3000 |
| T | Testing time | E/18 |

*Table 3. Halstead Complexity metrics, meanings and formulas.*

Taking as an example the code fragment in Figure 14, and taking as an operator (n1): main, (), {}, int, scanf, &, =, ,, +, /, ;, printf; and as operands (n2): x, y, z, avg, “%d %d %d”, 3, “avg = %d”

```
main()
{
  int x, y, z, avg;
  scanf(“%d %d %d”, &x, &y, &z);
  avg = (x+y+z)/3;
  printf(“avg = %d”, avg);
}
```

*Figure 8. Code example in C.*

Halstead metrics can be calculated like this:

$$n_1 = 12,\ n_2 = 7;\ n = 19$$

$$N_1 = 27,\ N_2 = 15,\ N = 42$$

$$N = 12*\log_2 12+7*\log_2 7 = 62.7$$

$$V = 42*\log_2 19 = 178.4$$

$$D = (12/2)+(15/7) = 12.85$$

$$E = 12.85*178.4 = 2292.44$$

$$T = 2292.44/18 = 127.357 \text{ seconds}$$

## 6.3. Tests

The testing phase of the software lifecycle is performed to determine whether the proposed design meets the initial set of objectives. It is aimed at detecting errors made in the previous stages in order to correct them. Of course, this should ideally be done before the end user encounters them. There are different types of tests, each with its own objectives, focus and methodologies.

## 6.3.1.

## Unit tests

The purpose of these is to verify the functionality and structure of each individual component that makes up the system. This type of test verifies that:

- The modules that make up the system are free of errors.
- All the main logical paths are executed correctly in each module of the application.
- All of the transactions performed by each module are executed correctly.

## 6.3.2.

## System tests

The purpose of these is global integration, verifying the correct functioning of all the interfaces between the different modules that make it up and with the rest of the information systems or tools with which it will communicate. The following list of tests are executed to do this:

- Functional tests, with the aim of ensuring that the system correctly performs all the functions detailed in the specifications provided by the system user.
- Integration tests to validate that the system communicates correctly with third-party applications. The man-machine interfaces are tested as part of this type of test.
- Performance tests to verify that the response times are within the intervals stipulated in the system specifications.
- Volume tests in which the system's operation will be monitored when it is working with large volumes of data. Tests are mainly performed in the modules.
- Overload tests to check the operation of the system when it is subjected to massive loads, with the aim of defining the maximum operating thresholds where the system operates below the established requirements.
- Backup tests to verify that the system can recover from failures, both physical and logical, without compromising data integrity.
- Ease-of-use tests to check the system's adaptability to the user's needs in terms of system usability.
- Operation tests to verify the correct implementation of the operating procedures, including planning and control of work, booting and rebooting the system.
- System security tests to verify system access control mechanisms.

### 6.3.3.

## Implementation tests

Implementation tests are carried out to verify the correct operation of all of the system's modules and functionalities, which are fully integrated in both the hardware and the software, in the final operating environment. So it will be the users who, from an operational point of view, accept the system once it has been implemented in the real operating environment. It should be emphasised that, at this point, system backup tests are also carried out to verify that its operation is not compromised by the existence of control and monitoring of the data safeguarding and recovery procedures.

### 6.3.4.

## Acceptance tests

A set of tests performed on the system to validate that it complies with the expected operation in terms of functionality and performance. These tests are defined and executed by the system users, as they are the people responsible for its acceptance.

### 6.3.5.

## Regression tests

Regression tests are performed whenever a new module or functionality is added to the project in order to verify that the implemented changes do not introduce undesired behaviour into the system. To perform these it is necessary to repeat the battery of tests defined above.

# 7.

# ISO certifications

The ISO, also known as the International Organization for Standardization, is an independent non-governmental international organisation. Through its members, it brings together experts to share knowledge and develop voluntary international standards covering aspects of technology and manufacturing to ensure quality, health, customer service, safety and data protection.

The main objective of ISO is to help companies optimise their processes in order to increase the quality of their services by providing a set of requirements and standards to follow. As seen throughout this document, a standard is described as an expert-endorsed method of best practice, whether for software development, risk assessment, or other activity.

International standards allow consumers to trust that products are reliable and of good quality. An ISO certificate is international recognition that offers companies the possibility of operating beyond national borders, enabling them to increase sales and income. ISO labels help to improve the image of companies, as they can demonstrate that they work according to international standards. For many buyers and customers, this is a sign that companies offer excellent products and services.

ISO standards seek to ensure quality, consistency and safety. There are multiple benefits for organisations that comply with these standards, such as trustworthiness, improved performance and quality, reduced risk, sustainability and innovation. Developing software under ISO standards ensures better code quality and robustness, reducing the likelihood of errors in production.

### ISOs applicable to code development

Some of the ISO standards for software development are explained below:

- **ISO 27000**: Information security
- **ISO 29119**: Software testing
- **ISO 25001**: Software product quality requirements and evaluation (SQuaRE)

[15] https://www.iso.org/

The ISO 27000 family includes standards for information security within an organisation. The primary purpose of this ISO is to protect the company's assets and improve its security practices.

ISO 27001 is the internationally recognised standard for information security management systems (ISMS). This consists of policies, processes, controls and other components involving people in the company as well as processes, property and technology infrastructure used by the organisation.

The primary purpose of this standard is ensuring that information and information processing facilities are secure and legal compliance is properly implemented and maintained across organisations and uses risk and opportunity management processes, risk assessments and risk treatments to mitigate and properly respond to security threats and incidents such as cybercrime, personal data breaches, damage, misuse and viral attacks. ISO 27001 includes controls for protecting personally identifiable information which assists with GDPR compliance, and as a method for achieving good data security.

ISO 29119 focuses on software testing. The main idea behind this ISO is that testing is the primary approach to risk mitigation and prevention. Therefore, all the standards follow the risk-based approach and encourage companies to focus on the most important functions.

ISO/IEC 25001:2014 provides requirements and recommendations for an organisation responsible for implementing and managing the systems and software product quality requirements specification and evaluation activities through the provision of technology, tools, experiences, and management skills.

Benefits:

- Ensuring a reduction in software failures after its implementation in production.
- Evaluating and monitoring the performance of the developed software product, ensuring that it will be able to deliver the results taking into account the time and resource constraints established.
- Ensuring that the developed software product respects the necessary levels for security features (confidentiality, integrity, authenticity, etc.).
- Verifying that the developed product can be put into production without compromising the rest of the systems and maintaining compatibility with the necessary interfaces.

# 8. References

[1] Index, T. I. O. B. E. (2018). Tiobe-the software quality company. TIOBE Index| TIOBE–The Software Quality Company [Electronic resource]. https://www. tiobe. com/tiobe-index/

[2] Fangohr, H. (2004). A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds) Computational Science - ICCS 2004. ICCS 2004. Lecture Notes in Computer Science, vol 3039. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-25944-2_157

[3] Baskova, O., Gatsenko, O., & Gordienko, Y. (2010). Enabling high-performance distributed computing to e-science by integration of 4th generation language environments with desktop grid architecture and convergence with global computing grid. In Proc. of Cracow Grid Workshop (CGW'10) (pp. 234-243).

[4] Ogala, J. O., & Ojie, D. V. (2020). Comparative analysis of c, c++, c# and java programming languages. GSJ, 8(5).

[5] A. Dinh, S. Miertschin, A. Young, and S. D. Mohanty, "A data-driven approach to predicting diabetes and cardiovascular disease with machine learning," BMC Medical Informatics and Decision Making, vol. 19, no. 1, Nov. 2019, doi: 10.1186/s12911-019-0918-5.

[6] Takeuchi, H., & Nonaka, I. (1986). The new product development game. Harvard business review, 64(1), 137-146.

[7] Sutherland, J. (1993). The Plan is the Problem!.

[8] Sutherland, J. V., & Schwaber, K. (1995). The SCRUM methodology. In Business object design and implementation: OOPSLA workshop.

[9] S. Al-Saqqa, S. Sawalha i H. AbdelNabi, «Agile Software Development: Methodologies and Trends,» International Journal of Interactive Mobile Technologies, pp. vol. 14, no. 11, 2020.

[10] H. Wickham, Advanced R, 2nd ed. Chapman and Hall/CRC, 2019.

[11] K. Reitz, The Hitchhiker's Guide to Python. O'Reilly Media, 2016. [Online]. Available: https://docs.python-guide.org/

[12] MISRA C, Guidelines for the use of the C language in critical systems. HORIBA MIRA Limited, 2020.

[13] MISRA C++, Guidelines for the use of the C++ language in critical systems. HORIBA MIRA Limited., 2020.

[14] "What is Quality Assurance (QA): Tutorial, Attributes, Components, Types." https://www.javatpoint.com/quality-assurance

[15] T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, doi: 10.1109/tse.1976.233837.

[16] T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai, "Software complexity analysis using Halstead metrics," May 2017, doi: 10.1109/icoei.2017.8300883.

# 9.
# Annexes

# ANNEX 1: Tidyverse Style Guide

## 1. Files

### 1.1 Names

File names should be meaningful and end in .R. Avoid using special characters in file names - stick with numbers, letters and "_".

### 1.2 Organisation

Giving a file a concise name helps with good organisation.

### 1.3 Internal structure

Use commented lines of "-" and "= " to break up your file into easily readable chunks.

## 2. Syntax

### 2.1 Object names

Variable and function names should use only lowercase letters, numbers, and "_". Use underscores (_) to separate words within a name (e.g. "day_1" instead of "day1").

### 2.2 Spacing

2.2.1 Commas

Always put a space after a comma, never before.

2.2.2 Parentheses

Do not put spaces inside or outside parentheses for regular function calls.

Place a space before and after () when used with if, for, or while.

Place a space after () used for function arguments.

2.2.3 Embracing

The embracing operator, {{ }}, should always have inner spaces to help emphasise its special behaviour.

2.2.4 Infix operators

The infix operators (== , + , - , <- , etc.) should always be surrounded by spaces.

2.2.5 Extra spaces

Adding extra spaces is ok if it improves alignment of = or <-.

### 2.3 Function calls

2.3.1 Named arguments

A function's arguments typically fall into two broad categories: one supplies the data to compute on; the other controls the details of computation. When you call a function, you typically omit the names of data arguments, because they are used so commonly. If you override the default value of an argument, use the full name.

2.3.2 Assignment

Avoid assignment in function calls.

**2.4 Control flow**

2.4.1 Code blocks

Curly braces, {}, define the most important hierarchy of R code. To make this hierarchy easy to see:

- { should be the last character on the line. The code must be on the same line as the opening brace {.
- The contents should be indented by two spaces.
- } should be the first character on the line.

2.4.2 If statements

- If used, else should be on the same line as }.
- & and | should never be used inside of an if clause because they can return vectors. Always use && and || instead.
- ifelse(x, a, b) is not a drop-in replacement for if (x) a else b.

2.4.3 Inline statements

It is ok to drop the curly braces for very simple statements that fit on one line, as long as they don't have side-effects. The return(), stop() or continue function calls should always go in their own {} block:

2.4.4 Implicit type coercion

Avoid implicit type coercion (e.g. from numeric to logical) in if statements:

2.4.5 Switch statements()

- Avoid position-based switch() statements.
- Each element should go on its own line.
- Elements that go to the next element must have a space after =.
- Provide a fall-through error, unless you have previously validated the input.

2.5 Long lines

Limit code to 80 characters per line.

**2.6 Semicolons**

Don't put ; at the end of a line, and don't use ; to put multiple commands on one line.

**2.7 Assignment**

Use <- , and not =, for assignment.

**2.8 Data**

2.8.1 Character vectors

Use " " not ' ' for quoting text. The only exception is when the text already contains double quotes and no single quotes.

2.8.2 Logical vectors

Prefer TRUE and FALSE over T and F.

**2.9 Comments**

Each line of a comment should begin with the comment symbol and a single space: #

## 3. Functions

### 3.1 Naming

Use verbs for function names following the instructions explained in point 2.1.

3.2 Long lines

There are two options if the function name and definition can't fit on a single line.

- Place each argument on its own line, and indent to match the opening ( of function.
- Place each argument on its own line. Place the first on the line after the opening of the function.

### 3.3 return()

Use return() for express returns only. Otherwise, rely on R to return the result of the last evaluated expression.

### 3.4 Comments

In code, use comments to explain the "why" not the "what" or the "how". Each line of a comment must begin with the comment symbol and a single space: #.

## 4. Pipes

### 4.1 Introduction

Use %>% to emphasise a sequence of actions, rather than the object that the actions are being performed on.

Avoid using the pipe when:

- You need to manipulate more than one object at a time. Reserve pipes for a sequence of steps applied to one primary object.
- There are meaningful intermediate objects that could be given informative names.

### 4.2 Whitespace

%>% should always have a space before it, and should usually be followed by a new line. After the first step, each line should be indented by two spaces. This structure makes it easier to add new steps (or rearrange existing steps) and harder to overlook a step.

### 4.3 Long lines

If the arguments to a function do not all fit on one line, put each argument on its own line and indent.

### 4.4 Short pipes

A one-step pipe can stay on one line. However, unless you plan to expand it later on, it is better to rewrite it as a regular function call.

### 4.5 No arguments

magrittr allows you to omit () on functions that do not have arguments. Avoid this feature.

**4.6 Assignment**

There are three acceptable forms of assignment:

- Variable name and assignment on separate lines
- Variable name and assignment on the same line
- Assignment at the end of the pipe with ->

# ANNEX 2: Google's R Style Guide

## 1. Files

**1.1 Names**

File names should be meaningful and end in .R. Avoid using special characters in file names - stick with numbers, letters and "_".

Everything must be documented.

**1.2 Organisation**

Giving a file a concise name helps with good organisation.

**1.3 Internal structure**

Use commented lines of "-" and "= " to break up your file into easily readable chunks.

## 2. Sintaxi

**2.1 Object names**

2.1.1 CamelCase

Function names have initial capital letters (CamelCase)

2.1.2 Dot at the beginning of a private function

The names of private functions should begin with a dot.

Spacing

**2.3. Spacing**

2.2.1 Commas

Always put a space after a comma, never before.

2.2.2 Parentheses

Do not put spaces inside or outside parentheses for regular function calls.

Place a space before and after () when used with if, for, or while.

Place a space after () used for function arguments.

2.2.3 Embracing

The embracing operator, {{ }}, should always have inner spaces to help emphasise its special behaviour.

2.2.4 Infix operators

Most infix operators (== , + , - , <-, etc.) should always be surrounded by spaces.

2.2.5 Extra spaces

Adding extra spaces is ok if it improves alignment of = or <-.

**2.3 Function calls**

2.3.1 Named arguments

A function's arguments typically fall into two broad categories: one supplies the data to compute on; the other controls the details of computation. When you call a function, you typically omit the names of data arguments, because they are used so commonly. If you override the default value of an argument, use the full name.

2.3.2 Assignment

Avoid assignment in function calls.

2.3.3 Do not use attach()

It is recommended not to use attach() as the possibility of creating errors is very high.

Control flow

**2.4. Control flow**

2.4.1 Code blocks

Curly braces, {}, define the most important hierarchy of R code. To make this hierarchy easy to see:

- { should be the last character on the line. The code must be on the same line as the opening brace {.
- The contents should be indented by two spaces.
- } should be the first character on the line.

2.4.2 If statements

- If used, else should be on the same line as }.
- & and | should never be used inside of an if clause because they can return vectors. Always use && and || instead.
- ifelse(x, a, b) it is not a drop-in replacement for if (x) a else b.

2.4.3 Inline statements

It is ok to drop the curly braces for very simple statements that fit on one line, as long as they don't have side-effects. The return(), stop() or continue function calls should always go in their own {} block:

2.4.4 Implicit type coercion

Avoid implicit type coercion (e.g. from numeric to logical) in if statements:

2.4.5 Switch statements

- Avoid position-based switch() statements.
- Each element should go on its own line.
- Elements that fall through to the following element should have a space after =.
- Provide a fall-through error, unless you have previously validated the input.

### 2.5 Long lines

Limit code to 80 characters per line.

### 2.6 Semicolons

Don't put ; at the end of a line, and don't use ; to put multiple commands on one line.

### 2.7 Assignment

Use <- , not =, for assignment.

### 2.8 Data

2.8.1 Character vectors

Use " " not " ' " for quoting text. The only exception is when the text already contains double quotes and no single quotes.

2.8.2 Logical vectors

Prefer TRUE and FALSE over T and F.

### 2.9 Comments

Each line of a comment should begin with the comment symbol and a single space: #

## 3. Funcions

### 3.1 Naming

Use verbs for function names following the directions given in point 2.

### 3.2 Long lines

There are two options if the function name and definition can't fit on a single line.

- Place each argument on its own line, and indent to match the opening ( of function.
- Place each argument on its own line. Place the first on the line after the opening of the function.

### 3.3 return()

Always use return().

### 3.4 Comments

In code, use comments to explain the "why" not the "what" or "how". Each line of a comment should begin with the comment symbol and a single space: #.

## 4. Pipes

### 4.1 Introduction

Use %>% to emphasise a sequence of actions, rather than the object that the actions are being performed on.

Avoid using the pipe when:

- You need to manipulate more than one object at a time. Reserve pipes for a sequence of steps applied to one primary object.
- There are meaningful intermediate objects that could be given informative names.

### 4.2 Whitespace

%>% should always have a space before it, and should usually be followed by a new line. After the

first step, each line should be indented by two spaces. This structure makes it easier to add new steps (or rearrange existing steps) and harder to overlook a step.

**4.3 Long lines**

If the arguments to a function do not all fit on one line, put each argument on its own line and indent.

**4.4 Short pipes**

A one-step pipe can stay on one line. However, unless you plan to expand it later on, it is better to rewrite it as a regular function call.

**4.5 No arguments**

magrittr allows you to omit () on functions that do not have arguments. Avoid this feature.

**4.6 Assignment**

There are two acceptable forms of assignment:

- Variable name and assignment on separate lines
- Variable name and assignment on the same line
- Assignment at the end of the pipe with -> is not acceptable

# ANNEX 3: Amazon AWS R Coding Style

The Amazon R Coding Style is based on Google's R Style Guide (see Annex 2) and "Advanced R, by Hadley Wickham", with a few tweaks as explained below.

### 1. Notation and naming

**1.1 File names**

File names should be meaningful and end in .R.

If files need to be run in sequence, prefix them with numbers.

**1.2 Object names**

Variable and function names should be in lowercase.

Use an underscore (_) to separate words within a name.

Variable names should be nouns and function names should be verbs.

Strive for names that are concise and meaningful.

Where possible, avoid using names of existing functions and variables. Doing so will cause confusion for the readers of your code.

## 2. Syntax

### 2.1 Spacing

Place spaces around all infix operators (=, +, -, <-, etc.). The same rule applies when using = in function calls. Always put a space after a comma, and never before.

Place a space before left parentheses, except in a function call.

Extra spacing (i.e. more than one space in a row) is ok if it improves alignment of equal signs or assignments (<-).

Do not place spaces around code in parentheses or square brackets.

### 2.2 Curly braces {}

An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line, unless it is followed by else.

### 2.3 Line length

Limit code to 80 characters per line. If you find yourself running out of room, this is a good indication that you should encapsulate some of the work in a separate function.

### 2.4 Indentation

When indenting your code, use two spaces. Never use tabs or mix tabs and spaces.

The only exception is if a function definition runs over multiple lines. In that case, indent the second line to where the definition starts.

### 2.5 Assignment

Use <-, not =, for assignment.

## 3. Organisation

### 3.1 Commenting Guidelines

Comment your code. Each line of a comment should begin with the comment symbol and a single space: #.

Comments should explain the “why”, not the “what” or the “how”.

Short comments can be placed after code preceded by two spaces, #, and then one space.

Use commented lines of - and = to break up your file into easily readable chunks.

### 3.2 Function definitions and calls

Function definitions should first list arguments without default values, followed by those with default values.

In both function definitions and function calls, multiple arguments per line are allowed. Line breaks are only allowed between assignments.

### 3.3 Function documentation

Functions should contain a comments section immediately below the function definition line. These comments should consist of:

- A one-sentence description of the function.
- A list of the function’s arguments, denoted by Args:, with a description of each (including the data type).

- And a description of the return value, denoted by “Returns:”.

Comments should be descriptive enough that a caller can use the function without reading any of the function’s code.

# ANNEX 4: PEP 8 – Style Guide for Python Code

## 1. Code lay-out

### 1.1 Indentation

Use 4 spaces per indentation level.

### 1.2 Tabs and spaces

Spaces are the preferred indentation method. Tabs should be used solely to remain consistent with code that is already indented with tabs. Python disallows mixing tabs and spaces for indentation.

### 1.3 Maximum line length

Limit all lines to a maximum of 79 characters.

### 1.4 Line break

A line break before the operator is recommended when writing formulas rather than a line break after the operator, although both options are still accepted.

### 1.5 Blank lines

- Surround top-level function and class definitions with two blank lines.
- Method definitions inside a class are surrounded by a single blank line.

### 1.6 Source file encoding

Code should always use UTF-8, and should not have an encoding declaration. In the standard library, non-UTF-8 encodings should be used only for test purposes.

### 1.7 Imports

- Imports should usually be on separate lines.
- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.
- Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured.

### 1.8 Module level dunder names

Module level “dunders” (i.e. names with two leading and two trailing underscores) should be placed after the module docstring but before any import statements.

## 2. String quotes

Single-quoted strings and double-quoted strings are the same. However, if you start with one type, you should continue with the chosen type.

## 3. Whitespace in expressions and statements

### 3.1 Pet Peeves

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces.
- Between a trailing comma and a following close parenthesis.
- Immediately before a comma, semicolon, or colon.
- However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied.
- Immediately before the open parenthesis that starts the argument list of a function call.
- Immediately before the open parenthesis that starts an indexing.
- More than one space around an assignment operator to align it with another.

### 3.2 Other recommendations

- Avoid trailing whitespace anywhere. Because it is usually invisible, it can be confusing.
- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >= , in, not in, is, is not), Booleans (and, or, not).
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.
- Function annotations should use the normal rules for colons and always have spaces around the “->” arrow if present.
- Do not use spaces around the “=” sign when used to indicate a keyword argument, or when used to indicate a default value for an unannotated function parameter.
- While sometimes it is okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements.

## 4. When to use trailing commas

Trailing commas are usually optional, except they are mandatory when making a tuple of one element. For clarity, it is recommended to surround the latter in (technically redundant) parentheses.

## 5. Comentaris

Put understandable code explanations. Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes.

### 5.1 Block comments

- Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code.
- Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).
- Paragraphs inside a block comment are separated by a line containing a single #.

### 5.2 Inline comments

Utilitzar els comentaris en línia amb moderació.

### 5.3. Documentation strings

Conventions for writing good documentation strings (a.k.a. “docstrings”) are immortalised in PEP 257[16].

## 6. Convencions de nomenclatura

### 6.1 Overriding principle

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

### 6.2 Names to avoid

Never use the characters “l” (lowercase letter el), “O” (uppercase letter oh) or “I” (uppercase letter eye) as single character variable names.

### 6.3 ASCII compatibility

Identifiers used in the standard library must be ASCII compatible as described in the policy section of PEP 3131[17].

### 6.4 Package and module names

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

### 6.5 Class names

Class names should normally use the CamelCase convention.

### 6.6 Exception names

You should use the suffix “Error” on your exception names (if the exception actually is an error).

### 6.7 Noms de variables globals

The conventions are about the same as those for functions.

### 6.8 Function and variable names

- Function names should be lowercase, with words separated by underscores as necessary to improve readability.
- Variable names follow the same convention as function names.

[16] https://peps.python.org/pep-0257/
[17] https://peps.python.org/pep-3131/

### 6.9 Function and method arguments

- Always use “self” for the first argument to instance methods.
- Always use “cls” for the first argument to class methods.

### 6.10 Method names and instance variables

- Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.
- Use one leading underscore only for non-public methods and instance variables.

### 6.11 Constants

Constants are usually defined on a module level and written in all capital letters with underscores separating words.

### 6.12 Design for inheritance

Always decide whether a class’s methods and instance variables (collectively: “attributes”) should be public or non-public. If in doubt, choose non-public; it is easier to make it public later than to make a public attribute non-public.

## 7. Programming recommendations

- Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such).
- Comparisons to singletons like “None” should always be done with “is” or is “not”, never the equality operators.
- Use “is not” operator rather than “not … is”.
- Always use a “def” statement instead of an assignment statement that binds a lambda expression directly to an identifier.
- Use exception chaining appropriately.
- When catching operating system errors, prefer the explicit exception hierarchy introduced in Python 3.3 over introspection of “errno” values.
- For all try/except clauses, limit the try clause to the absolute minimum amount of code necessary. This avoids masking bugs.
- When a resource is local to a particular section of code, use a “with” statement to ensure it is cleaned up promptly and reliably after use.
- Context managers should be invoked through separate functions or methods whenever they do something other than acquire and release resources.
- Be consistent in “return” statements. All return statements in a function should return an expression.
- Use ‘’.startswith() and ‘’.endswith() instead of string slicing to check for prefixes or suffixes. startswith() and endswith() are cleaner and less error prone.
- Object type comparisons should always use isinstance() instead of comparing types directly.
- For sequences, (strings, lists, tuples), use the fact that empty sequences are false.

- Don’t write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors will trim them.
- Do not compare Boolean values to True or False using “==”.

# ANNEX 5: Google Python Style Guide

This Python style guide written by Google is based on the official Python PEP8 guide. If a rule is not stated below, that does not mean that it does not exist, but that the rule in PEP8 prevails (see Annex 4)

## 1. Python style rules

### 1.1 Semicolons

Do not terminate your lines with semicolons. Do not use semicolons to put two statements on the same line.

### 1.2 Line length

The maximum line length is 80 characters.

### 1.3 Parentheses

Use parentheses sparingly.

It is fine, though not required, to use parentheses around tuples. Do not use them in return statements or conditional statements unless using parentheses for implied line continuation or to indicate a tuple.

### 1.4 Indentation

Indent your code blocks with 4 spaces.

Never use tabs or a mix of tabs and spaces. Implied line continuation should align wrapped elements vertically or use a hanging 4-space indent. In this case there should be nothing after the opening parenthesis or bracket on the first line.

### 1.5 Blank lines

- Two blank lines between top-level definitions, be they function or class definitions.
- One blank line between method definitions and between the docstring of a class and the first method.
- No blank line following a “def” line.
- Use single blank lines as you judge appropriate within functions or methods.
- Blank lines need not be anchored to the definition.

### 1.6 Whitespace

- Follow standard typographic rules for the use of spaces around punctuation.
- No whitespace inside parentheses, brackets or braces.
- No whitespace before or after a comma, period or semicolon, except at the end of the line.
- No whitespace between :, #, =, etc to align code
- Surround binary operators with a single space on either side for assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), and Booleans (and, or, not).

### 1.7 Shebang line

Most .py files do not need to start with a #! line. Start the main file of a program with #!/usr/bin/env python3 (to support virtualenvs) or #!/usr/bin/python3 per PEP-394[18]. This line is used by the kernel to find the Python interpreter, but is ignored by Python when importing modules. It is only necessary on a file intended to be executed directly.

### 1.8 Comments and docstrings

Be sure to use the right style for module, function, method docstrings and inline comments.

### 1.9 Strings

Use an f-string, the % operator, or the “format” method for formatting strings, even when the parameters are all strings.

### 1.10 Files, sockets, and similar stateful resources

Explicitly close files and sockets when done with them. This rule naturally extends to closeable resources that internally use sockets, such as database connections, and also other resources that need to be closed down in a similar fashion.

### 1.11 TODO comments

- Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect.
- The purpose is to have a consistent TODO format that can be searched to find out how to get more details.
- A TODO is not a commitment that the person referenced will fix the problem. Thus when you create a TODO with a username, it is almost always your own username that is given.
- If the TODO describes future changes, make sure that you include either a specific date or a specific event.

### 1.12 Imports formatting

Imports should be on separate lines; there are exceptions for “typing” and “collections.abc” imports.

### 1.13 Statements

Generally only one statement per line.

### 1.14 Getters and setters

Getter and setter functions (also called accessors and mutators) should be used when they provide a meaningful role or behaviour for getting or setting a variable’s value. In particular, they should

[18] https://peps.python.org/pep-0394/

be used when getting or setting the variable is complex or the cost is significant, either currently or in a reasonable future.

**1.15 Naming**

Function names, variable names, and filenames should be descriptive; avoid abbreviation. In particular, do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

**1.16 Main**

In Python, pydoc, as well as unit tests, require modules to be importable. If a file is intended to be used as an executable, its main functionality should be in a main function, and your code should always check “if __name__ == ‘__main__’” before executing your main program, so that it is not executed when the module is imported.

**1.17 Function length**

Prefer small and focused functions.

**1.18 Type annotations**

There are general rules such as: only annotate “self” or “cls” if it is necessary for proper type information. If any other variable or a returned type should not be expressed, use “Any”. You are not required to annotate all the functions in a module.

Follow the rules set out above concerning spaces and indentation.

# ANNEX 6: MISRA C:2012 Rules and Directives

Description of the columns from left to right: list of rules and directives, category, and description. The rules and guidelines are categorised by levels according to need: Required, advisory and mandatory.

| RULES AND DIRECTIVES | CATEGORY | DESCRIPTION |
|---|---|---|
| D.1.1 | Required | Any implementation-defined behaviour on which the output of the program depends shall be documented and understood |
| D.2.1 | Required | All source files shall compile without any compilation errors |
| D.3.1 | Required | All code shall be traceable to documented requirements |
| D.4.1 | Required | Run-time failures shall be minimised |
| D.4.2 | Advisory | All usage of assembly language should be documented |
| D.4.3 | Required | Assembly language shall be encapsulated and isolated |
| D.4.4 | Advisory | Sections of code should not be “commented out” |
| D.4.5 | Advisory | Identifiers in the same name space with overlapping visibility should be typographically unambiguous |
| D.4.6 | Advisory | Typedefs that indicate size and signedness should be used in place of the basic numerical types |
| D.4.7 | Required | If a function returns error information, then that error information shall be tested |
| D.4.8 | Advisory | If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden |
| D.4.9 | Advisory | A function should be used in preference to a function-like macro where they are interchangeable |
| D.4.10 | Required | Precautions shall be taken in order to prevent the contents of a header file being included more than once |
| D.4.11 | Required | The validity of values passed to library functions shall be checked |
| D.4.12 | Required | Dynamic memory allocation shall not be used |
| D.4.13 | Advisory | Functions which are designed to provide operations on a resource should be called in an appropriate sequence |
| D.4.14 | Required | The validity of values received from external sources shall be checked |
| R.1.1 | Required | The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation’s translation limits |
| R.1.2 | Advisory | Language extensions should not be used |
| R.1.3 | Required | There shall be no occurrence of undefined or critical unspecified behaviour |
| R.1.4 | Required | Emergent language features shall not be used |
| R.2.1 | Required | A project shall not contain unreachable code |

| RULES AND DIRECTIVES | CATEGORY | DESCRIPTION |
|---|---|---|
| R.2.2 | Required | There shall be no dead code |
| R.2.3 | Advisory | A project should not contain unused type declarations |
| R.2.4 | Advisory | A project should not contain unused tag declarations |
| R.2.5 | Advisory | A project should not contain unused macro declarations |
| R.2.6 | Advisory | A function should not contain unused label declarations |
| R.2.7 | Advisory | There should be no unused parameters in functions |
| R.3.1 | Required | The character sequences /* and // shall not be used within a comment |
| R.3.2 | Required | Line-splicing shall not be used in // comments |
| R.4.1 | Required | Octal and hexadecimal escape sequences shall be terminated |
| R.4.2 | Advisory | Trigraphs should not be used |
| R.5.1 | Required | External identifiers shall be distinct |
| R.5.2 | Required | Identifiers declared in the same scope and name space shall be distinct |
| R.5.3 | Required | An identifier declared in an inner scope shall not hide an identifier declared in an outer scope |
| R.5.4 | Required | Macro identifiers shall be distinct |
| R.5.5 | Required | Identifiers shall be distinct from macro names |
| R.5.6 | Required | A typedef name shall be a unique identifier |
| R.5.7 | Required | A tag name shall be a unique identifier |
| R.5.8 | Required | Identifiers that define objects or functions with external linkage shall be unique |
| R.5.9 | Advisory | Identifiers that define objects or functions with internal linkage should be unique |
| R.6.1 | Required | Bit-fields shall only be declared with an appropriate type |
| R.6.2 | Required | Single-bit named bit fields shall not be of a signed type |
| R.7.1 | Required | Octal constants shall not be used |
| R.7.2 | Required | A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type |
| R.7.3 | Required | The lowercase character “l” shall not be used in a literal suffix |
| R.7.4 | Required | A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualified char” |
| R.8.1 | Required | Types shall be explicitly specified |
| R.8.2 | Required | Function types shall be in prototype form with named parameters |
| R.8.3 | Required | All declarations of an object or function shall use the same names and type qualifiers |

| RULES AND DIRECTIVES | CATEGORY | DESCRIPTION |
|---|---|---|
| R.8.4 | Required | A compatible declaration shall be visible when an object or function with external linkage is defined. |
| R.8.5 | Required | An external object or function shall be declared once in one and only one file |
| R.8.6 | Required | An identifier with external linkage shall have exactly one external definition |
| R.8.7 | Advisory | Functions and objects should not be defined with external linkage if they are referenced in only one translation unit |
| R.8.8 | Required | The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage |
| R.8.9 | Advisory | An object should be defined at block scope if its identifier only appears in a single function |
| R.8.10 | Required | An inline function shall be declared with the static storage class |
| R.8.11 | Advisory | When an array with external linkage is declared, its size should be explicitly specified |
| R.8.12 | Required | Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique |
| R.8.13 | Advisory | A pointer should point to a const-qualified type whenever possible |
| R.8.14 | Required | The restrict type qualifier shall not be used |
| R.9.1 | Mandatory | The value of an object with automatic storage duration shall not be read before it has been set |
| R.9.2 | Required | The initialiser for an aggregate or union shall be enclosed in braces { } |
| R.9.3 | Required | Arrays shall not be partially initialised |
| R.9.4 | Required | An element of an object shall not be initialised more than once |
| R.9.5 | Required | Where designated initialisers are used to initialize an array object the size of the array shall be specified explicitly |
| R.10.1 | Required | Operands shall not be of an inappropriate essential type |
| R.10.2 | Required | Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations |
| R.10.3 | Required | The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category |
| R.10.4 | Required | Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category |
| R.10.5 | Advisory | The value of an expression should not be cast to an inappropriate essential type |
| R.10.6 | Required | The value of a composite expression shall not be assigned to an object with wider essential type |
| R.10.7 | Required | If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type. |
| R.10.8 | Required | The value of a composite expression shall not be cast to a different essential type category or a wider essential type |
| R.11.1 | Required | Conversions shall not be performed between a pointer to a function and any other type. |
| R.11.2 | Required | Conversions shall not be performed between a pointer to an incomplete type and any other type. |
| R.11.3 | Required | A cast shall not be performed between a pointer to object type and a pointer to a different object type |
| R.11.4 | Advisory | A conversion should not be performed between a pointer to object and an integer type |

| RULES AND DIRECTIVES | CATEGORY | DESCRIPTION |
|---|---|---|
| R.11.5 | Advisory | A conversion should not be performed from pointer to void into pointer to object |
| R.11.6 | Required | A cast shall not be performed between pointer to void and an arithmetic type |
| R.11.7 | Required | A cast shall not be performed between pointer to object and a non-integer arithmetic type |
| R.11.8 | Required | A cast shall not remove any const or volatile qualification from the type pointed to by a pointer |
| R.11.9 | Required | The macro NULL shall be the only permitted form of integer null pointer constant |
| R.12.1 | Advisory | The precedence of operators within expressions should be made explicit |
| R.12.2 | Required | The righthand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand |
| R.12.3 | Advisory | The comma operator should not be used |
| R.12.4 | Advisory | Evaluation of constant expressions should not lead to unsigned integer wrap-around |
| R.12.5 | Mandatory | The sizeof operator shall not have an operand which is a function parameter declared as “array of type” |
| R.13.1 | Required | Initialiser lists shall not contain persistent side effects |
| R.13.2 | Required | The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders |
| R.13.3 | Advisory | A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator |
| R.13.4 | Advisory | The result of an assignment operator should not be used |
| R.13.5 | Required | The right hand operand of a logical && or \|\| operator shall not contain persistent side effects |
| R.13.6 | Mandatory | The operand of the sizeof operator shall not contain any expression which has potential side effects |
| R.14.1 | Required | A loop counter shall not have essentially floating type |
| R.14.2 | Required | A for loop shall be well-formed |
| R.14.3 | Required | Controlling expressions shall not be invariant |
| R.14.4 | Required | The controlling expression of an if statement and the controlling expression of an iteration- statement shall have essentially Boolean type |
| R.15.1 | Advisory | The goto statement should not be used |
| R.15.2 | Required | The goto statement shall jump to a label declared later in the same function |
| R.15.3 | Required | Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement |
| R.15.4 | Advisory | There should be no more than one break or goto statement used to terminate any iteration statement |
| R.15.5 | Advisory | A function should have a single point of exit at the end |
| R.15.6 | Required | The body of an iteration-statement or a selection-statement shall be a com-pound-statement |
| R.15.7 | Required | All if … else if constructs shall be terminated with an else statement |
| R.16.1 | Required | All switch statements shall be well-formed |

| RULES AND DIRECTIVES | CATEGORY | DESCRIPTION |
|---|---|---|
| R.16.2 | Required | A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement |
| R.16.3 | Required | An unconditional break statement shall terminate every switch-clause |
| R.16.4 | Required | Every switch statement shall have a default label |
| R.16.5 | Required | A default label shall appear as either the first or the last switch label of a switch statement |
| R.16.6 | Required | Every switch statement shall have at least two switch-clauses |
| R.16.7 | Required | A switch-expression shall not have essentially Boolean type |
| R.17.1 | Required | The features of <stdarg.h> shall not be used |
| R.17.2 | Required | Functions shall not call themselves, either directly or indirectly |
| R.17.3 | Mandatory | A function shall not be declared implicitly |
| R.17.4 | Mandatory | All exit paths from a function with non-void return type shall have an explicit return statement with an expression |
| R.17.5 | Advisory | The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements |
| R.17.6 | Mandatory | The declaration of an array parameter shall not contain the static keyword between the [ ] |
| R.17.7 | Required | The value returned by a function having non-void return type shall be used |
| R.17.8 | Advisory | A function parameter should not be modified |
| R.18.1 | Required | A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand |
| R.18.2 | Required | Subtraction between pointers shall only be applied to pointers that address elements of the same array |
| R.18.3 | Required | The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object |
| R.18.4 | Advisory | The +, -, += and -= operators should not be applied to an expression of pointer type |
| R.18.5 | Advisory | Declarations should contain no more than two levels of pointer nesting |
| R.18.6 | Required | The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist |
| R.18.7 | Required | Flexible array members shall not be declared |
| R.18.8 | Required | Variable-length array types shall not be used |
| R.19.1 | Mandatory | An object shall not be assigned or copied to an overlapping object |
| R.19.2 | Advisory | The union keyword should not be used |
| R.20.1 | Advisory | #include directives should only be preceded by preprocessor directives or comments |
| R.20.2 | Required | The ‘ , ” or \ characters and the /* or // character sequences shall not occur in a header file name |
| R.20.3 | Required | The #include directive shall be followed by either a <filename> or “filename” sequence |
| R.20.4 | Required | A macro shall not be defined with the same name as a keyword |

| RULES AND DIRECTIVES | CATEGORY | DESCRIPTION |
|---|---|---|
| R.20.5 | Advisory | #undef should not be used |
| R.20.6 | Required | Tokens that look like a preprocessing directive shall not occur within a macro argument |
| R.20.7 | Required | Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses |
| R.20.8 | Required | The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1 |
| R.20.9 | Required | The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1 |
| R.20.10 | Advisory | The # and ## preprocessor operators should not be used |
| R.20.11 | Required | A macro parameter immediately following a # operator shall not immediately be followed by a ## operator |
| R.20.12 | Required | A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators |
| R.20.13 | Required | A line whose first token is # shall be a valid preprocessing directive |
| R.20.14 | Required | All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related |
| R.21.1 | Required | #define and #undef shall not be used on a reserved identifier or reserved macro name |
| R.21.2 | Required | A reserved identifier or macro name shall not be declared |
| R.21.3 | Required | The memory allocation and deallocation functions of <stdlib.h> shall not be used |
| R.21.4 | Required | The standard header file <setjmp.h> shall not be used |
| R.21.5 | Required | The standard header file <signal.h> shall not be used |
| R.21.6 | Required | The Standard Library input/output functions shall not be used |
| R.21.7 | Required | The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used |
| R.21.8 | Required | The library functions abort, exit, getenv and system of <stdlib.h> shall not be used |
| R.21.9 | Required | The library functions bsearch and qsort of <stdlib.h> shall not be used |
| R.21.10 | Required | The Standard Library time and date functions shall not be used |
| R.21.11 | Required | The standard header file <tgmath.h> shall not be used |
| R.21.12 | Advisory | The exception handling features of <fenv.h> should not be used |
| R.21.13 | Mandatory | Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF |
| R.21.14 | Required | The Standard Library function memcmp shall not be used to compare null terminated strings |
| R.21.15 | Required | The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types |
| R.21.16 | Required | Arguments to the standard library memcmp function must point to a pointer, assigned type, Boolean, or enum type. |
| R.21.17 | Mandatory | Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters |
| R.21.18 | Mandatory | The size_t argument passed to any function in <string.h> shall have an appropriate value |

| RULES AND DIRECTIVES | CATEGORY | DESCRIPTION |
|---|---|---|
| R.21.19 | Mandatory | The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type |
| R.21.20 | Mandatory | The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function |
| R.21.21 | Required | The Standard Library function system of <stdlib.h> shall not be used |
| R.22.1 | Required | All resources obtained dynamically by means of Standard Library functions shall be explicitly released |
| R.22.2 | Mandatory | A block of memory shall only be freed if it was allocated by means of a Standard Library function |
| R.22.3 | Required | The same file shall not be open for read and write access at the same time on different streams |
| R.22.4 | Mandatory | There shall be no attempt to write to a stream which has been opened as read-only |
| R.22.5 | Mandatory | A pointer to a FILE object shall not be dereferenced |
| R.22.6 | Mandatory | The value of a pointer to a FILE shall not be used after the associated stream has been closed |
| R.22.7 | Required | The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF |
| R.22.8 | Required | The value of errno shall be set to zero prior to a call to an errno-setting-function |
| R.22.9 | Required | The value of errno shall be tested against zero after calling an errno-setting-function |
| R.22.10 | Required | The value of errno shall only be tested when the last function to be called was an errno-setting-function |

TIC Salut Social

Programa Salut i IA

Generalitat de Catalunya