**Good Data Scientists Write Good Code**

Tips on how to be nice to yourself and your colleagues when developing code for data products



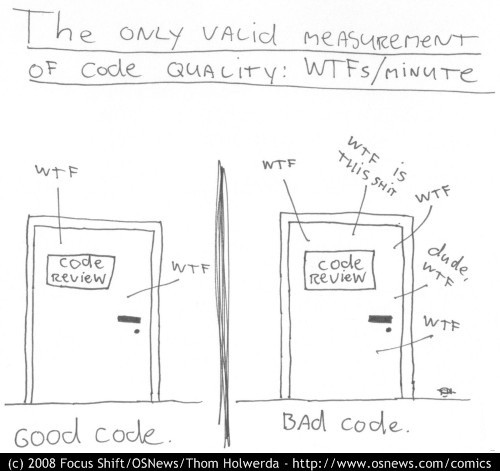
Photo by [Quino Al](https://unsplash.com/@quinoal?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

**Data products are software applications**

As Data Scientists, we build *data products*, i.e. products that use data to solve real-world problems. Data products can take many different shapes and forms depending on the problem at hand and include static reports, real-time dashboards, interactive web applications, Machine Learning-based web services, etc. What unites these types of products is that building them involves writing code, e.g. in Python, R, Julia, and/or a dialect of SQL. Because of this, the majority of data products are, in essence, software applications albeit of different complexity.

The complexity of code behind a given data product depends not only on the nature of that product but also on the stage of its development. [CRISP-DM](https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining) is a well-known framework that describes what happens at different stages of Data Science projects. CRISP-DM includes six major stages: (i) business understanding, (ii) data understanding, (iii) data preparation, (iv) modelling, (v) evaluation, and (vi) deployment. As the project advances, requirements for the quality and robustness of the underlying codebase typically increase. Moreover, Data Science projects often require well-coordinated efforts from participants as diverse as business stakeholders, other Data Scientists, Data Engineers, IT Operations specialists, etc. Thus, to make the whole process as smooth and effective as possible and to minimise the associated business risks, a good Data Scientist should (among [many other things](https://www.forbes.com/sites/quora/2017/06/15/what-are-the-top-five-skills-data-scientists-need/)) be able to write good code.

But what makes a “good code” exactly? Perhaps the most concise answer to this question can be found in the following illustration that opens the popular “[Clean Code](https://www.pearson.com/en-us/subject-catalog/p/clean-code-a-handbook-of-agile-software-craftsmanship/P200000009044/9780132350884)” book by R. C. Martin:



The image is reproduced with permission from OSNews.com

To reduce [WTF](https://dictionary.cambridge.org/dictionary/english/wtf)s/minute in Data Science code, we can follow some of the best practices developed over decades in the field of [Software Engineering](https://en.wikipedia.org/wiki/History_of_software_engineering). In [my work](http://nextgamesolutions.com/) as Data Scientist and consultant, I found the following best practices and principles to be particularly relevant and useful:

* it takes minimal time and cost for new team members to join the project;
* the code is written in a modular way and is version-controlled and unit-tested;
* the application is fully configurable and contains no hard-coded values that control its execution;
* the application is portable between execution environments;
* the application is scalable without changing the tooling, architecture or development practices;
* no code gets deployed in production without a peer review;
* there is a monitoring infrastructure that allows one to track and understand how the application behaves in production.

Presented below are brief commentaries on these best practices in the context of Data Science, as well as some recommendations on how to implement them. This article might be of particular interest to Data Scientists in the early stages of their careers. However, I believe seasoned professionals and Data Science managers can find useful nuggets of information here as well.

**Writing a perfect piece of code in one go is almost impossible, and that’s OK**

Having clearly formulated requirements in Data Science projects is uncommon. For example, business stakeholders may want to predict a certain quantity using a Machine Learning model but they would rarely be able to say anything about the acceptable uncertainty for that prediction. Many other unknowns may further hinder and slowdown Data Science projects, for example:

* Can the problem at hand be solved with data in the first place?
* Which data would be helpful to solve that problem?
* Can Data Scientists access that data?
* Are the available data of sufficient quality, and do they contain a sufficiently strong “signal”?
* How much would it cost to set up and maintain a feed with the required data in the future?
* Are there any regulatory constraints for using certain types of data?

As a result, Data Science projects often take a lot of time and resources, become highly iterative (see CRISP-DM mentioned above), and may fail altogether. Given the highly risky nature of Data Science projects, especially in their early stages, it doesn’t make much sense to write production-grade code from Day 1. Instead, it’s useful to approach code development more pragmatically, similar to how it’s done in software development projects.

[One approach](https://wiki.c2.com/?MakeItWorkMakeItRightMakeItFast) to software development that I find particularly relevant suggests thinking about the evolution of the application codebase in terms of the following stages: “*make it work, make it right, make it fast*”. In the context of Data Science, these stages can be interpreted as follows:

* *Make it work*: develop a prototype solution for the business problem at hand in order to analyse that solution, learn from it, and decide if further development and deployment are justified. For instance, this could involve quickly building a predictive model on a limited sample of data using default hyperparameters. The code at this stage doesn’t have to be “pretty”.
* *Make it right*: progressing to this stage would be justified if the prototype development work showed a promising result and a decision was made to develop and deploy a full-fledged data product. The prototype code now gets thrown away, and a new, production-grade, code is written following the corresponding requirements and best practices.
* *Make it fast*: this stage is likely to be reached after running the deployed application in production for a while. Observing the behaviour of a data product “in the wild” often reveals all sorts of computational inefficiencies. While the product itself may be delivering the business value as expected, such inefficiencies could reduce the overall ROI of the project by incurring unnecessary costs (e.g., due to the computational costs on a cloud platform). When this happens, one may need to go back to the application codebase and try to optimise it.

Let us now dive deeper into each of these ideas and see how the code quality requirements change as the project evolves.

**I. Making it work**



Image by [Bernd](https://pixabay.com/users/momentmal-5324081/) from [Pixabay](https://pixabay.com/photos/cart-wood-middle-ages-spokes-2581608/)

**Build a prototype first**

Data Science, just like any other Science, is all about “figuring stuff out”, finding what works and what doesn’t for a given problem. This is why at the beginning of a project it is important to *start small* and build a *prototype* first. According to [Wikipedia](https://en.wikipedia.org/wiki/Prototype),

*“A prototype is an early sample, model, or release of a product built to test a concept or process.”*

The last part of this definition is particularly important: we build prototypes to analyse and learn from them as cheap as possible so that we can quickly decide whether further development and deployment are justified [ROI-wise](https://towardsdatascience.com/return-on-investment-for-machine-learning-1a0c431509e). When used properly, a prototype can save substantial amounts of time, money, and suffering early in the development cycle.

The actual form that prototypes take in Data Science projects depends on the problem being solved. Examples include, but are not limited to:

* a piece of exploratory analysis demonstrating the value of available data;
* a quick-n-dirty predictive model, whose performance metrics help to understand whether the problem at hand can be solved with data;
* a web application running locally on a static dataset and built to collect early end-user feedback.

**A prototype code is a throw-away code**

The code of a prototype solution needs to be treated as *disposable code*. As a Data Scientist leading the project or as a Project Manager, you must make this *very clear* to your stakeholders. To cite one of my favourite books, “[The Pragmatic Programmer](https://www.amazon.co.uk/Pragmatic-Programmer-journey-mastery-Anniversary/dp/0135957052/)” (Hunt & Hunt 2019),

*“It’s easy to become misled by the apparent completeness of a demonstrated prototype, and project sponsors or management may insist on deploying the prototype (or its progeny) if you don’t set the right expectations. Remind them that you can build a great prototype of a new car out of balsa wood and duct tape, but you wouldn’t try to drive it in rush-hour traffic!”*

Prototypes of data products do *not* actually have to be written as code. To move quickly, a prototype can (and sometimes maybe even should) be built using [low-code or no-code Data Science tools](https://towardsdatascience.com/no-code-low-code-ai-new-business-models-and-future-of-data-scientists-a536beb8d9e3). However, when a prototype *is*written as code, there are several do’s and don’t’s that one should keep in mind.

**What’s OK to do when building a prototype as code**

Since the prototype code is disposable, it:

* doesn’t have to be “pretty” or optimised for computational speed as the speed of development is way more important at this stage;
* doesn’t have to be documented to the same level as a production-grade code (however, there should be enough comments and/or Markdown-based notes to understand what the code does and to ensure its reproducibility);
* doesn’t have to be version-controlled (although setting up version control at the beginning of any project is always a good idea);
* it’s OK to have hard-coded values (but *not*the sensitive ones, such as passwords, API keys, etc.);
* can be written and stored in a Jupyter Notebook or similar media (e.g., [R Markdown Notebook](https://rmarkdown.rstudio.com/lesson-10.html)) instead of being organised into a library of functions and/or a collection of production-ready scripts.

**What’s *not*OK to do when building a prototype as code**

Although the prototype code is disposable, there is always a chance that the project will advance to the next stage which involves developing a full-fledged data product. Thus, one should always make their best effort at writing prototype code that resembles production-grade code as much as possible. This can substantially speed up the subsequent development process. At a bare minimum, here is what’s *not*OK to do when building a prototype as code (see below for further discussion):

* using cryptic or abbreviated names for variables and functions;
* mixing code styles (e.g., randomly using camelCase and snake\_case to name variables and functions in languages like Python or R, or using uppercased and lowercased command names in SQL);
* not using comments in the code at all;
* having sensitive values (passwords, API keys, etc.) exposed in-code;
* not storing notebooks or scripts with the prototype code for future reference.

**II. Making it right**



Image by [Emslichter](https://pixabay.com/users/emslichter-1377910/) from [Pixabay](https://pixabay.com/illustrations/citroen-type-c-5cv-cabriolet-6655053/)

If the prototyping stage showed positive results, the project can move on to developing a production-ready solution. The process of making a data product available to its human users or other systems is referred to as *deployment*. Deployment of data products often brings about a whole lot of hard requirements, including but not limited to:

* project-specific [SLA](https://en.wikipedia.org/wiki/Service-level_agreement)s (for instance, sufficiently low response time from an API that serves model predictions, high availability and concurrency for web applications, etc.);
* infrastructure to deploy and run the application in a fully automated and scalable way ([DevOps](https://en.wikipedia.org/wiki/DevOps)/[MLOps](https://en.wikipedia.org/wiki/MLOps));
* infrastructure to robustly deliver high-quality input data when and if required by the application;
* infrastructure to monitor the operational (e.g., CPU load) and product-related metrics (e.g., the accuracy of predictions);
* continuous operational support for business-critical applications, etc.

Many of these requirements are to be covered by the Data Engineering, MLOps, and DevOps teams. Moreover, as the project progresses, the role these engineering teams play becomes increasingly more important. However, Data Scientists also have to “make things right” on their end. Let’s see what that implies in terms of code quality.

**Use descriptive variable names**

Can you make a guess as to what x, y, and z mean in the below piece of code?

z = x / y^2

Without knowing the context that this code applies to, it’s essentially impossible to say what each of the three variables represents. But what if we instead were to re-write that line of code as follows:

body\_mass\_index = body\_mass / body\_height^2

Now it’s crystal clear what the code does — it calculates the [body mass index](https://en.wikipedia.org/wiki/Body_mass_index) by dividing the body mass by the body height squared. Moreover, this code is fully [*self-documenting*](https://en.wikipedia.org/wiki/Self-documenting_code)— there is no need to provide any additional comments as to what it’s calculating.

It’s very important to realise that production-grade code is written not for ourselves but mainly for *other people* — those, who at some point will review it, contribute to it, or maintain it going forward. Using *descriptive names*for variables, functions, and other objects will help to significantly reduce the *cognitive effort*required to understand what the code does and will make the code more maintainable.

**Use a consistent coding style**

The cognitive effortrequired to read and understand a piece of code can be further reduced if that code is formatted according to a standard. All major programming languages have their official *style guides***.**It doesn’t matter much which style exactly a team of developers chooses because ultimately it’s all about consistency. However, it really helps if the team picks one style and adheres to it, and it then becomes the collective responsibility of every team member to enforce the use of the adopted style. Examples of the code styles commonly used for Data Science languages include [PEP8](https://www.python.org/dev/peps/pep-0008/) (Python), [tidyverse](https://style.tidyverse.org/) (R), and [SQL Style Guide](https://www.sqlstyle.guide/) by Simon Holywell (SQL).

To write stylistically consistent code without having to think about the rules of the chosen style, use an [*integrated development environment*](https://en.wikipedia.org/wiki/Integrated_development_environment) (IDE), such as [VSCode](https://code.visualstudio.com/), [PyCharm](https://www.jetbrains.com/pycharm/), or [RStudio](https://www.rstudio.com/products/rstudio/). The popular [Jupyter Notebooks](https://jupyter.org/) can be convenient for prototyping, however, they are *not* meantfor writing production-grade code as they [lack most of the features](https://docs.google.com/presentation/d/1n2RlMdmv1p25Xy5thJUhkKGvjtV-dkAIsUXP-AL4ffI/) a professional IDE offers (e.g., code highlighting and auto-formatting, real-time type checking, navigating to function and class definitions, dependency management, integration with version control systems, etc.). Code executed from a Jupyter Notebook is also prone to many [security issues](https://www.helpnetsecurity.com/2017/01/26/jupyter-notebooks-security-hole/) and all sorts of unpredictable behaviours.

If, for some reason, your favourite IDE cannot auto-format code according to a given style, use a dedicated library to do that, such as [pylint](https://pypi.org/project/pylint/) or [flake8](https://pypi.org/project/flake8/) for Python, [lintr](https://cran.r-project.org/web/packages/lintr/index.html) or [styler](https://cran.r-project.org/web/packages/styler/index.html) for R, and [sqlfluff](https://www.sqlfluff.com/) for SQL.

**Write modular code**

Modular code is code that is broken down into small independent parts (e.g., functions), each doing *one thing* and one thing only. Organising code this way makes it much easier to maintain, debug, test, share with other people, and ultimately helps to write new programs faster.

When designing and writing functions, stick to the following best practices:

* *Keep it short*. If you find your function contains tens of lines of code, consider splitting that code even further.
* *Make the code easy to read and comprehend*. In addition to using descriptive names, this can be achieved by avoiding highly specialised constructs of the programming language used in the project (e.g., intricate list comprehensions, excessively long method chains, or decorators in Python), unless they speed up the code significantly.
* *Minimise the number of function arguments*. It’s not a hard rule, and there are exceptions, but if the function you write has more than 3–5 arguments, it’s probably doing too many things, so consider splitting its code further still.

The Internet is full of examples of how one could modularise Python code. For R, there is an excellent and freely available book “[R Packages](https://r-pkgs.org/)”. Many useful recommendations of this sort for Data Scientists can also be found in [Laszlo Sragner](https://www.linkedin.com/in/laszlosragner/)’s [blog](https://hypergolic.co.uk/blog/).

**Never hard-code sensitive information**

To be secure, production code must *never*expose any sensitive information in the form of hard-coded constants. Examples of sensitive information include resource handles to databases, user passwords, credentials for third-party services (e.g., cloud services or social media platforms), etc. A good litmus test for whether your code correctly factored out such variables is whether it could be open-sourced at any moment without compromising any credentials. The [Twelve-Factor App](https://12factor.net/) methodology recommends storing sensitive information in [*environment variables*](https://en.wikipedia.org/wiki/Environment_variable). One convenient and secure way of working with such variables involves storing them as key-value pairs in a special [.env file](https://dev.to/jakewitcher/using-env-files-for-environment-variables-in-python-applications-55a1), which never gets committed to the app’s remote code repository. Libraries for reading such files and their content exist in both [Python](https://github.com/theskumar/python-dotenv) and [R](https://github.com/gaborcsardi/dotenv).

**Explicitly declare all dependencies in a manifest file**

It’s common to see data products, whose codebase is dependent on tens of specialised libraries. The code may work well on the developer’s local machine, where all of these dependencies are installed and function correctly. However, transferring applications to production environments often breaks things if the code dependencies are not managed properly. To avoid such problems, one must declare all dependencies, completely and exactly, via a [*dependency declaration manifest*](https://12factor.net/dependencies). This manifest takes different forms depending on the language, e.g. the [requirements.txt](https://pip.pypa.io/en/stable/user_guide/#requirements-files) file in Python applications or the [DESCRIPTION](https://r-pkgs.org/Metadata.html) file in R packages.

Furthermore, it is recommended to use a *dependency isolation* tool during the application’s execution to avoid interference from the host system. Commonly used examples of such tools in Python are [venv](https://docs.python.org/3/library/venv.html), [virtualenv](https://virtualenv.pypa.io/en/latest/), and [conda environments](https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html). A similar tool — [renv](https://rstudio.github.io/renv/index.html) — exists also for R.

**Use a single, version-controlled code repository**

The code of a production-intended data product must be *version-controlled*(VC)andstored in a remote repository accessible by other project members. [Git](https://git-scm.com/) is perhaps the most commonly used VC system, and nowadays Data Scientists developing production-ready data products are expected to be familiar at least with its basic principles. These basics, as well as more advanced techniques, can be learnt from numerous resources, such as the “[Pro Git](https://git-scm.com/book/en/v2)” book, “[Happy Git and GitHub for the UseR](https://happygitwithr.com/)” book, or the “[Oh Shit, Git!?!](https://ohshitgit.com/)” website. The most commonly used platforms for remote code repository management are [GitHub](https://github.com/), [Bitbucket](https://bitbucket.org/), and [GitLab](https://about.gitlab.com/).

VC is important for a number of reasons, including the following:

* it enables collaboration between project members;
* when things break, it allows one to roll back to a previous working version of the application;
* it enables an automated [continuous integration and deployment](https://www.redhat.com/en/topics/devops/what-is-ci-cd);
* it creates full transparency, which can be particularly important for audit purposes in regulated industries.

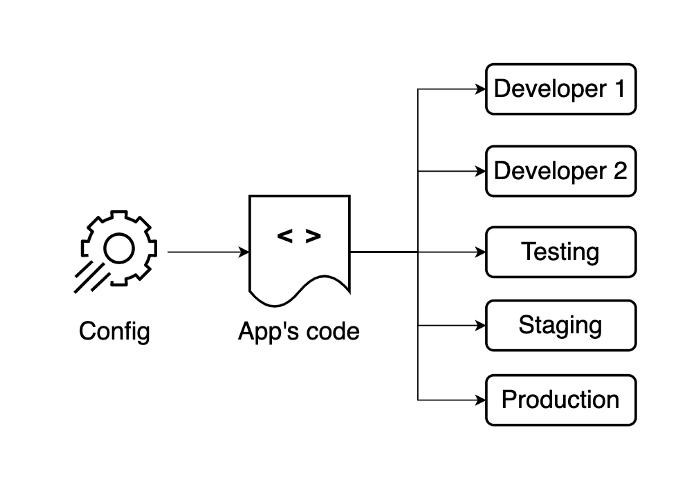
The codebase of a given application is to be stored in a *single repository*, and different applications should not share the same pieces of code. If you find yourself re-using the same code in different applications or projects, it’s time to factor out that reusable code into a *separate library*with its own codebase.

**Use a config file to store non-sensitive application parameters**

According to the [Twelve-Factor App](https://12factor.net/config) methodology,

*“An app’s config is everything that is likely to vary between*[*deploys*](http://deploys/)*(staging, production, developer environments, etc.).”*

This includes all sorts of credentials, resource URIs, passwords, and also Data Science-specific variables, such as model hyperparameters, values to impute missing observations, names of the input and output datasets, etc.



An app’s config allows for running the same app with different settings depending on the environment. Image by the author

It is almost guaranteed that a production-intended data product will need a config. One example of why that might be the case is when running the application in a development environment doesn’t require as much computational power as in production. Instead of hard-coding the respective computational resource values (e.g., AWS EC2 instance parameters) into the application’s main code, it makes much more sense to treat those values as configuration parameters.

Config parameters can be stored in [many different ways](https://stackoverflow.com/questions/6198372/most-pythonic-way-to-provide-global-configuration-variables-in-config-py). The Twelve-Factor App methodology [recommends](https://12factor.net/config) storing them in environment variables. However, some applications can have too many parameters to keep track of in the form of environment variables. In such cases, it’s more sensible to place *non-sensitive* parameters in a dedicated version-controlled *config file* and use environment variables to store sensitive parameters only.

The most common formats for config files are JSON and YAML. YAML files are human-readable and are thus often preferable. YAML config files can be easily read and parsed in both Python (e.g., using the [anyconfig](https://github.com/ssato/python-anyconfig) or [pyyaml](https://pyyaml.org/wiki/PyYAML) libraries) and R (e.g., using the [yaml](https://github.com/vubiostat/r-yaml) or [config](https://github.com/rstudio/config) packages).

**Equip your application with a logging mechanism**

*Logs*are a stream of time-ordered events collected from the output streams of all active processes and [*backing services*](https://12factor.net/backing-services) of an application. These events are commonly written to a text file on the server’s disk, with one event per line. They are being generated continuously as long as the application is operating. The purpose of logs is to provide visibility into the behaviour of a running application. As a result, logs are extremely useful for detecting failures and bugs, alerting, calculating the time taken by various tasks, etc. As with other software applications, it’s strongly recommended that Data Scientists inject logging mechanisms into data products that are to be deployed in production. There are easy-to-use libraries for doing that (e.g., [logging](https://docs.python.org/3/howto/logging.html) in Python and [log4r](https://github.com/johnmyleswhite/log4r) in R).

It should be noted that Machine Learning-based applications have additional monitoring needsbecause of the *non-deterministic nature*of the respective algorithms. Examples of such additional metrics include indicators of the quality of input and output data, data drifts, the accuracy of predictions, and other, application-specific, quantities. This, in turn, requires the project team to spend additional time and effort to build the (often complex) infrastructure to operate and support deployed models. Unfortunately, there are currently no standard approaches for building such an infrastructure — the project team will have to choose what’s essential for their specific data product and decide on the best way of implementing the respective monitoring mechanisms. In certain cases, one can also opt-in for a commercial monitoring solution, such as those offered by [AWS SageMaker](https://docs.aws.amazon.com/sagemaker/latest/dg/model-monitor.html), [Microsoft Azure ML](https://learn.microsoft.com/en-us/azure/machine-learning/v1/how-to-monitor-datasets?tabs=python), [etc](https://neptune.ai/blog/ml-model-monitoring-best-tools).

**Test your code for correctness**

No code should go into production without confirming that it does what it’s supposed to be doing. The best way to confirm that is to write a battery of use case-specific *tests*, which can then be automatically run at critical time points (for example, at each [pull request](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests)). This kind of testing is often referred to as “[*unit testing*](https://en.wikipedia.org/wiki/Unit_testing)”, where “*unit refers to low-level test cases written in the same language as the production code, which directly access its objects and members”*([Hunt & Hunt 2019](https://www.amazon.co.uk/Pragmatic-Programmer-journey-mastery-Anniversary/dp/0135957052/)). Unit tests are usually written for individual functions but can also cover classes and higher-level modules.

There are two main approaches to writing automated tests. One of them is when a function or some other piece of functionality is written first, and then the tests for it are written. The other approach is known as [*test-driven development*](https://en.wikipedia.org/wiki/Test-driven_development) and implies that one first writes tests according to the expected behaviour of the unit of functionality, and only then codes up the unit itself. There are pros and cons to each of these methods, and which one is “better” is a bit of a philosophical discussion. In practice, it doesn’t really matter which method to use as long as the team sticks to a consistent way of working.

All major programming languages have dedicated libraries to create automated code tests. Examples include [unittest](https://docs.python.org/3/library/unittest.html) and [pytest](https://docs.pytest.org/en/7.0.x/) for Python and [testthat](https://testthat.r-lib.org/) for R.

An important concept in unit testing is [*code coverage*](https://en.wikipedia.org/wiki/Code_coverage)— the percentage of code that is covered by automated tests. This metric does *not*have to be equal to 100% (moreover, in practice it’s often difficult to reach a high code coverage, especially in large projects). Some of the general recommendations in this regard are as follows:

* write a test for every situation when you are tempted to type something into a print statement to see if the code works as expected;
* avoid testing simple code that you are confident will work;
* always write a test when you discover a bug.

Code coverage can be automatically calculated using a variety of tools, such as the [coverage](https://coverage.readthedocs.io/) library in Python or [covr](https://covr.r-lib.org/) in R. There also exist specialised commercial platforms that can calculate code coverage, track it for each project over time, and visualise the results. A good example is [Codecov](https://about.codecov.io/).

At a bare minimum, unit tests should be run locally on the developer’s machine and supplied in the project’s repository so that other project members can re-run them. However, “the right way” to do this nowadays is to run tests fully automatically on a dedicated automation server (e.g., [Jenkins](https://www.jenkins.io/)) or using other continuous integration tools (e.g., [GitHub Actions](https://github.com/features/actions), [GitLab CI](https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/), [CircleCI](https://circleci.com/), etc.).

**Make sure at least one person peer-reviews your code**

A piece of code may well be bugs-free and cleanly formatted, but there are almost always certain aspects that can be improved even further. For example, the code author can use certain language constructs that make it difficult for other people to read that code and understand what it does (e.g., long method chains, intricate list comprehensions, etc.). In many cases, it makes sense to simplify these constructs for readability’s sake. Spotting parts of the code that might be improved requires a “fresh pair of eyes”, and a peer reviewer can help with that. The reviewer is typically someone from the team, who is familiar with the project or directly contributes to it.

Peer review is also important when making changes in the codebase that may significantly impact the end users.

Overall, the purpose of peer review is to simplify the code, make obvious improvements in terms of its execution speed, as well as identify functional errors (i.e. spot situations when the code executes just fine but in fact is not doing what it’s supposed to be doing).

The best way to organise code review and automatically track its progress is by assigning a reviewer at pull request using the respective functionality of a remote repository management platform. All major VC platforms offer such functionality.

**Document things, religiously**

Production-grade code *must* be documented. Without documentation, it will be very difficult to maintain code in the long term, distribute it efficiently, and onboard new team members quickly and smoothly. The following recommendations will help with creating a well-documented codebase:

* When writing classes, methods or pure functions, use [*docstrings*](https://www.python.org/dev/peps/pep-0257/)for Python or a similar language-specific mechanism if you are working with other languages. Using these standard mechanisms is very helpful because: (i) they automatically produce help files that one can call from the console to understand what a given class or function is doing and what arguments it expects; (ii) they make it possible to use special libraries (e.g., [sphinx](https://www.sphinx-doc.org/en/master/), [pydoc](https://docs.python.org/3/library/pydoc.html) or [pdoc](https://pdoc3.github.io/pdoc/) for Python, [roxygen2](https://github.com/r-lib/roxygen2) in combination with [pkgdown](https://pkgdown.r-lib.org/) for R) in order to generate web-ready documentation (i.e. in the form of HTML files) that can be easily shared with other people within an organisation or publicly.
* When writing scripts, (i) provide a *comment at the top*and explain briefly the purpose of the script; (ii) provide *additional comments*in places that might be challenging to understand otherwise.
* In the project’s remote repository, provide a [README file](https://www.makeareadme.com/)that explains all the relevant details about that repository (including a description of its structure, name of its maintainer(s), installation instructions, and, if applicable, links to external supporting documentation for the entire project).

**III. Making it fast**



Image by [Emslichter](https://pixabay.com/users/emslichter-1377910/) from [Pixabay](https://pixabay.com/photos/mercedes-benz-190-sl-cabriolet-3342783/)

A deployed application may reveal various computational inefficiencies and bottlenecks. However, before rushing into the [*code refactoring*](https://en.wikipedia.org/wiki/Code_refactoring), it is helpful to recall the following “[*rules of optimisation*](https://wiki.c2.com/?RulesOfOptimization)”:

1. *First Rule of Optimisation*: Don’t
2. *Second Rule of Optimisation*: Don’t… yet
3. *Profile Before Optimising*

In other words, it is always a good idea to make sure the problem is real before investing your precious time trying to fix it. This is done by “*profiling*” (as per Rule 3 above), which in Data Science projects usually means two things:

1. measuring the performance of the code of interest;
2. considering the intended optimisation work in the context of the entire project’s ROI and the business value it generates.

Just because a piece of code could theoretically be sped up doesn’t mean it should. The performance gain can turn out to be too small to warrant any additional work. And even if the expected performance gain is potentially significant, the project team has to assess whether making the proposed change is justified from the economic point of view. The proposed optimisation work will only make sense if the expected ROI of the entire project is positive and sufficiently large. Thus, the final decision is to be made by all the project members involved — it’s not only up to a Data Scientist anymore to conclude that the optimisation work is justified.

**Conclusion**

Data products are, in essence, software applications of different complexity. Yet, Data Scientists, who develop such applications, usually don’t have formal training in software development. Luckily, though, mastering the core principles of software development is not that difficult. This article provides a brief overview of such principles in the context of Data Science and will, hopefully, help Data Scientists with writing higher-quality code for their projects.