DHSC Coding Principles

Matthew Malcher

Contents

Introduction

These principles are designed with the aim of improving coding standards and consistency within the department.

Adoption of these principles should improve quality, facilitate collaboration and enable effective QA of code. The principles are not language specific. This is to maximise uptake and provide a uniform set of values across languages.

The principles are designed to be achievable by all DHSC analysts producing code. Each Principle is flexible and has multiple levels:

- **Must** - you aren't finished until your code has met this standard.
- **Should** - do this unless you are ready to justify not doing so.
- **Could** - things you can do to improve your code beyond the base standard.

Principles

Chapter 1

Use Version Control

It's important to version control your code so you and your collaborators can track changes over time, trace back errors, and retrieve old versions. Doing version control well can be tricky, fortunately, there are tools to help you.

You Must - Use version control and follow guidance on coding in the open.

You Should - Use standard tools (Git & GitHub) to help you version control code.

You Could - Use Git & GitHub collaboratively and agree a version control pattern with your team.

1.1 Git - (program for version control)

Git is the most popular way to version control code. You can tell Git to keep track of a folder full of code (a repository or 'repo'). As you make changes, Git identifies them and asks which should be kept (or 'committed'), and for a comment identifying why. It uses this to maintain a log of all the changes. This log can be used to restore old versions.

1.2 GitHub - (service for storing code version controlled using git)

GitHub is the most popular place to store git repositories. Storing your repository on GitHub has some key benefits:

- It acts as a backup of your code & its history
- Other people can see your code, download it and collaborate with you.

As such, using a service like GitHub is a good way to ensure that your code is available to your team, and that knowledge is transferred.

1.3 Using Git & GitHub Effectively

To make effective use of Git and GitHub:

- Follow the GDS Git Style Guide
- Commit code in small chunks, as often as possible, and write a descriptive comment with each commit. This makes your repository much more useful in the future and is called doing 'atomic commits'.
- Commit changes using an account which is specific to you so that it is clear in future who changed what.
- Define a version control pattern which explains how you use git as a team. This could include how you use branches, how you merge changes and how to handle pull requests.

1.4 Code in the Open

GitHub repositories can be *open* (shared with anyone) or *private* (shared only with selected collaborators).

The Technology Code of Practice encourages us to 'be open and use open source'. As such, **code should be open unless there is a good reason for it not to be.**

Good reasons for code to be private might be that it includes classified material, or information on an unreleased policy. GDS have published guidance on this for both external, and internal use.

1.5 Manual Version Control

It is *possible* to manually keep track of your code by systematically keeping copies of your code as you modify it and maintaining a corresponding log of all changes to the code. The log should include: What has changed, why, and a link to the relevant copy of the files. It is recommended to use version control tools (git) instead of this manual process.

Chapter 2

Write Easy to Read Code

Keep your code easy to read by using sensible names and a consistent style.

You Must - Use meaningful names and follow the DHSC adopted style for your language.

You Should - Use a linter or code formatter to ensure that your code conforms to the style guide.

You Could - Review your code with colleagues to make ensure your names and style promote understanding.

2.1 Style Guides

Most languages have several available style guides, which define a set of conventions to produce clean and consistently formatted code. Your style guide will define things like:

- How to use indentation and spacing
- Line length
- Add comment blocks at the start of your code
- Favour named indexes and iterators

DHSC has adopted a single style for each language to promote consistency. See style guides and Linters for more.

2.2 Meaningful Names

Names convey meaning, naming functions & variables well can remove the need for a comment and make life easier for other readers. This includes your future self!

Find a balance: avoid meaningless names like `obj` or `foo`; but don't put an entire sentence in a variable name.

Use single-letter variables only where the letter represents a well-known mathematical property (e.g. e = mcˆ2), or where their meaning is otherwise clear.

2.3 Name Formats

Follow your style guide and format your names consistently (i.e. using camelCase or snake_case).

Chapter 3

Correct, Clear, Fast & Concise - In That Order

Write code with your colleagues in mind. They need your code to work correctly, and they will have to understand and check it before they can derive benefit from it being fast or concise. Choose clarity over cleverness – just because you can do it in one line doesn't mean you should.

You Must - Write code with these priorities in mind.

You Should - Document design choices you have made to balance clarity, speed and concision.

You Could - Use profiling tools to understand resource usage and refactor to improve clarity and performance.

3.1 Correct

The first priority should always be that the code is correct. See the demonstably correct principle.

3.2 Clear

It is more valuable to have code that other analysts can quickly understand, than code which runs a little quicker. Your work needs to be quality assured - so at least one other person will need to understand what you have written!

Clarity is made up of many components (e.g. comments, easy to read code and data structures). But don't overlook the clarity of your approach to the problem.

Have you done something which will be impossible for someone else to check? Is there a more effective way to do it?

3.3 Fast

You may find that you have produced code which takes some time to run. If you expect to run it many times, *then* its time to think about how you could make things faster. Don't fall into the trap of optimising before you need to. For most languages there are profiling tools you can use to understand resource usage when you need to.

3.4 Concise

Keeping the amount of code you use to achieve a goal at a minimum can often be a good thing. There is less code to go wrong or debug, less to explain, style and document. But, remember that concision is less important than correctness, clarity and speed. Don't make it shorter than it needs to be, and think of the cost to clarity and flexibility.

Chapter 4

Write Flexible Code

Often your code will be adapted, repurposed, or applied to something new. Preparing for this upfront and writing flexible code can save time later.

This does **not** mean trying to solve every conceivable problem up-front, or writing general purpose functions for problems you may never encounter. Instead focus on making your code easy to change when needed:

You Must - Break your code into chunks, each with a clear purpose and don't repeat yourself. Think about, and document the way your code might break with different inputs.

You Should - Include input validation in your code.

You Could - Implement and test thorough error handling. Consider writing and sharing general purposes 'tool' code, especially if you solve a problem someone else might have.

4.1 Break up your code and don't repeat yourself

Don't Repeat Yourself is a common idea in programming. Instead of writing a bit of code several times to repeat an operation, you would re-use the same code by turning it into a function (or some other re-useable chunk - a method or subroutines).

You should look to break up your code into chunks rather than working in a single long script or program. Each chunk of code should have a clear purpose. You can then use call on these chunks of code to build your analysis.

There are many reasons to do this:

- It's easier to see the structure of your code.

- It's easier to make changes to your code.
- It's easier to understand what has changed and what the effect is.
- You only have to document each chunk once.

There are some common coding styles for breaking down a program into chunks.

4.2 Error Handling

Its likely your code will be asked to do something you didn't design it for, and it will break.

You do know however what your code *should* be doing, what the input is expected to look like, and the properties of the output. You also probably have an idea of what your would *like* it to do when it doesn't work correctly.

You can use error or exception handling to control the errors and warnings that your code produces. This can allow you to and choose: * Where your code breaks - by adding validation to a function input * How your code breaks - by providing alternate code to be run in the event of an error. * What it tells a user - by providing your own error messages which guide the user to a solution

There are automated techniques such as fuzzing available to check how your code and error handling responds to different inputs.

Chapter 5

Comment Effectively

The function of your code should be clear to another team member. Comments can be invaluable for capturing why a bit of code exists and what it is doing. However, they can also get in the way if there are too many of them or they are inconsistent with the code. Comments aren't just for other people, imagine if you return to your code in a years time – would you understand it?

You Must - Comment your code.

You Should - Think about *why* you are leaving comments, what to capture, and what belongs elsewhere.

You Could - Review old code you have written - are the comments helpful? what would you include next time?

5.1 Comment Style

Use comments judiciously and look to your style guide for advice on how to comment.

5.2 What to Capture

Ask yourself: "will I understand this code in 3 years time?"

Comments should bridge the gap between the documentation and the code. They should be written in plain English and describe the logic and purpose of each chunk of code; i.e. where it fits in and why its there.

5.2.1 What not to Capture

You don't need to describe everything you are doing with the code in a comment. Someone reading the code should be literate in the language. Commenting extensively line by line makes it likely that code and comments will get out of sync when you go back and make changes.

There should be separate documentation for high level questions such as the structure or logic of the analysis. Don't write your plan or QA notes in the comments.

Don't store large chunks of alternate code in the comments. Leave that to your version control system.

5.3 Alternatives

If you find yourself writing extensive comments, or writing more comments than code, consider changing format. There are options such as Jupyter Notebooks, and Rmarkdown for combining analysis and prose.

Chapter 6

Document Your Work

Commented, and well written code can go a long way to documenting how a project works. However for higher level context it will always be useful to have some separate documentation. Code without documentation won't be very useful later!

You Must - Produce documentation in line with Aqua book guidance.

You Should - Assemble documentation as you code.

You Could - Use document generation tools to produce documentation.

6.1 Aqua Book Guidance

The Aqua book contains guidance on documentation that should be in place as part of a quality assurance process for any analysis. This includes code based analysis. See Aqua Book Guidance for Technical Documentation for more.

6.2 Document as you Go

Don't fall into the trap of assuming documentation is something which is produced at the end. The best time to put together the documentation is as you are planning or doing the work - while it is fresh in your mind. This also means that switching projects wont result in undocumented work.

6.3 Use a Documentation Generator

There are popular tools for generating documentation from your code and comments. These lighten the load of producing and publishing good documentation, and encourage you to produce thorough documentation.

There are many documentation generators. Doxygen or Roxygen are recommended.

Chapter 7

Be Demonstrably Correct

We need to be confident in the outputs we provide. Just because something is done with code doesn't mean it is more likely to be answering the right question, using the right inputs, or doing the calculation correctly. As with any analysis, the defence against this is having a clear and robust audit trail which can *demonstrate* that our output is correct.

You Must - Hold your code to the same standard as regular analysis and record evidence demonstrating it produces the right output.

You Should - Use version control to unambiguously link QA to code and outputs and construct automated tests to provide confidence that changes don't break things.

You Could - Make a fully automated reproducible analytical pipeline (RAP).

7.1 QA Applies to Code

Code is not exempt from Quality Assurance processes. As with any other analysis you need to record evidence that your code is:

- doing the right thing,
- structured in a sensible way
- using valid inputs
- producing a sensible answer

7.2 Testing Frameworks

Your code and analysis will grow and evolve. You wont have time to QA every version, and it can be tricky to keep track of which bits of QA have been made obsolete due to new or changed code.

There are frameworks which help you construct and run tests on units of your code. These can be a good way to demonstrate that code is working correctly as you update it. See Testing Frameworks for more details.

7.3 Version Control Integration

Having unit tests, and QA is good. Ideally however you can tie a particular result to a particular version of the QA'd and tested code. You *could* do this manually, by keeping the code for each set of outputs.

Using git for version control makes this process easy. You can:

- Make a commit to the repository with a note like: `output for XYZ on dd/mm/yy` so you can identify the version used to produce outputs in future.
- Use tools such as gitpython or git2r to include the git commit hash which identifies the current version of the code in the output. This can then later be retrieved from your git repository.

7.4 Reproducible Analytical Pipelines

Once you have some QA'd, version controlled and test covered code, the biggest source of error will be the manual steps performed by the analyst running it. You can eliminate a lot of this, see Reproducible Analytical Pipelines.

Chapter 8

Use Sensible Defaults

There are basic tasks we all do. We should do these in the same way to make collaboration easier.

It is not true to say that there is only one right way of tackling a problem, or that one way is the best for all cases. However, collaboration and QA are easier if there is *consistency* in our approaches. That is usually more important than using the absolute best method.

You Must - Be aware of the defaults, understand why we have them and follow them unless you can explain how the benefits of an alternative approach outweigh those of consistency.

You Should - Help define what the defaults should be, and actively participate in discussion and debate to keep them up to date and relevant.

You Could - Proactively review and compare the defaults used

8.1 General Defaults

- Use 'Tidy' data. See the tidy data principle!
- Use git for version control of code (rather than SVN, Mercurial etc). See the version control principle
- Use a standardised template for your type of project, this will help people find things. See the reproducible principle

8.2 Language Specific Defaults:

In addition to the general principles, there are some sensible defaults which are language specific:

- R
- Python

Chapter 9

Be Reproducible

Within DHSC analysis is used to enable evidence based decision making. A piece of evidence which you cannot rely on being able to reproduce is not much good. There are many reasons your code may not work correctly if someone else tries to run it, as such there is a responsibility to understand how to make your analysis simple and easy to reproduce.

You Must - Keep track of what you have done and document it unambiguously so that someone else can recreate it.

You Should - Write portable code, in a standard project structure so that it is *easy* for someone else to run it.

You Could - Turn your code into a package / library / module, learn and promote RAP techniques, or use containers to achieve reproducibility.

9.1 Documenting for reproducibility

To be able to reproduce your analysis a colleague may need the following:

- The right copy of the code
- The right versions of any dependencies (i.e. libraries used in the code)
- The platform on which code is run
  - operating system
  - folder structure
  - machine specifications
- The source data, or details of how to get it.

At the most basic level, documenting all of these will go a long way to making your analysis reproducible. It might not make it *easy* to reproduce however.

9.2 Portability

There are some simple thing you can do to improve the chance that your code runs on other computers:

- Use relative paths, not absolute paths. (Wikipedia - Absolute and Relative Paths).
- Use a standard and consistent structure for organising your work. See Projects and Environments for more details.

9.3 Reproducible Analytical Pipelines

There is a government community dedicated to the production of reproducible analysis. See Reproducible Analytical Pipelines for more.

9.4 Packages and Modules

Most languages have a standard structure which is used to share code and documentation with other people. You will likely have used code in this structure (libraries / packages / modules) when performing your analysis. Typically these structures include documentation, information about dependencies, and tests.

There is no reason you can't use the same approach to sharing your analysis! See Packages for more.

9.5 Containers & Virtual Machines

Containers allow you to manage the whole environment which a bit of code runs in. They are powerful but perhaps more technically involved than packaging your code or using project structures to manage your environment. See Containers for more.

Chapter 10

Use Appropriate and Tidy Data

Programming languages offer many different structures for working with data. Using the right one for the job will often make a task easier, and decrease your chance of getting it wrong. At DHSC we are working with data represented in tables. For this type of work, using 'Tidy' data is usually the best approach.

You Must - Know what 'Tidy' data is, and understand why it is valuable.

You Should - Be familiar with the data types and structures available to you and ensure that you use the right ones.

You Could - Think about relationships between datasets, design schemas and store data in an efficient way.

10.1 Tidy Data?

A dataset is a collection of values, usually either numbers (if quantitative) or strings (if qualitative). Values are organised in two ways Every value belongs to a *variable* and an *observation*:

- A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units.
- An observation contains all values measured on the same *unit* (like a person, or a day, or a race) across attributes.

The majority of data we work with comes in rectangles. For this data to be tidy, ensure that: 1. Each variable forms a column. 2. Each observation forms a row. 3. Each type of observational unit forms a table.

For more see the section on *Tidy Data* in R for Data Science or the original paper.

Use tidy data structures as part of your work. You should attempt to convert incoming data into tidy format as quickly as possible. Any data that is output that may be used in other projects should be in tidy format as well as any other required formats.

10.2 Data Types and Structures

Data *types* are the basic units which your language uses to store data, things like integers, doubles, strings and logical data. Typically you are working with data frames, arrays, matricies or lists. These hold multiple items of data in a data *structure*.

Different types and structures are used for different things, and have different capabilities. To be effective, know about the data types and structures available to you and use the right ones for the job!

10.2.1 R

The R Programming for Data Science book has a good section on the 'Nuts and Bolts' of R which covers types and structures. For more about the different data structures a good resource is the Advanced R book.

10.2.2 Python

For a list of python datatypes see the:

- Python Documentation
- Pandas Documentation

10.3 Schema

The R for data science book has a nice section on relational data.

Appendix A

Aqua Book Guidance for Technical Documentation:

Pages 42-43 of the Aqua book contains guidance on documentation that should be in place as part of a quality assurance process for any analysis. The scope of the Aqua book is wider than code, however the definition of technical documentation is useful and repeated here:

All analysis should have documentation for the user, even if that 'user' is just the analyst leading the analysis.

This is to ensure that they have captured sufficient material to assist them if the analysis is revisited in due course. For analysis that is more likely to be revisited or updated in the future, documentation should be provided to assist a future analyst and should be more comprehensive.

This documentation should include: * a summary of the analysis including the context to the question being asked, * what analytical methods were considered, * what analysis was planned and why, * what challenges were encountered and how they were overcome * and what verification and validation steps were performed.

In addition, guidance on what should be considered if the analysis is to be revisited or updated is beneficial.

Appendix B

Coding styles

There are many different styles for breaking a program down into chunks:

- Functional Programming where programs are defined as a series of functions with defined inputs and outputs.
- Object Orientated Programming where programs are built on the concept of objects, which can contain data and procedures or methods for interacting with other objects.
- Procedural Programming where you provide a series of steps or procedures or routines to be carried out, which could be broken down further into steps or subroutines

Appendix C

Containers / Docker

Docker - is a containerisation platform, which lets you reproduce environments with a wider scope than just the packages present. With Docker you can manage the entire environment from the operating system and network up (including any packages). You can even use tools such as docker-compose to manage groups of containers relative to one another.

Appendix D

DHSC Sensible Defaults for Python

The following are the DHSC sensible defaults for Python:

- Use Python 3
- Use pandas for data analysis
  - Use `loc` and `iloc` to write to data frames
- Use Altair for basic data visualisation
- Use Scikit Learn for machine learning
- Use SQLAlchemy and pandas for database interactions, rather than writing your own SQL

Appendix E

DHSC Sensible Defaults for R

The following are the DHSC sensible defaults for R:

- Always work in a project
- Default to packages from the Tidyverse, because they have been carefully designed to work together effectively as part of a modern data analysis workflow. More info can be found here: R for Data Science by Hadley Wickham. For example:
- Prefer tibbles to data.frames
- Use ggplot2 rather than base graphics
- Use the pipe `%>%` rather than nesting function calls, but not always e.g. see here.
- Prefer `purrr` to the `apply` family of functions. See here
- If possible, build an R Package to share and document your code. Packages are the fundamental unit of reproducible R code.

Appendix F

Packages and Modules

Python - Packaging Projects

R - R Packages

Appendix G

Project Structure

Most languages offer tools and templates for different types of project work. Typically these include most of the following components:

- Source Data
- Code
- Outputs
- Environment / Dependencies
- Documentation

By following a standard template for these components you can take advantage of workflow tools provided by your IDE which make it easier to:

- Version Control your work
- Organise your code and source data
- Refactoring and improving your code
- Producing documentation
- Control your environment and dependencies

All of these things are good for sharing or collaborating with others.

G.1 R

The dominant IDE for R is Rstudio, which provides Rstudio Projects:

- Guide to Using Projects

Packages

Documentation for usethis

renv & packrat

G.2 Python

Python has many different options, all supported by different IDEs and tools

Anaconda & Conda Projects: Getting started with Conda

Pycharm Projects: Pycharm Projects

venv

Appendix H

Reproducible Analytical Pipelines

Automating the steps used to create a your report or output is a good way to avoid the human errors that manual intervention will introduce.

This is the focus of the Reproducible Analytical Pipelines community. The RAP community has curated the following resources:

- RAP Website
- Udemy Course

H.1 Automating Pipelines

You can do this by:

- Designing a final output which can be created without manual intervention.
- Making sure your code is broken down into chunks which do discrete tasks in your pipeline, for example:
  - gathering
  - cleaning
  - processing and modelling
  - reporting and visualisation
- Taking advantage of tools which keep track of the interactions between your data and code. These tools can then re-run the required bits of your pipeline automatically as you update, correct and improve it.
  - GNU Make is the classic tool and is language agnostic, but perhaps not user friendly.

  - The `drake` package is designed for analysis pipelines in R.
  - The `doit` package can do the same and more for python. The python wiki also has a list of build tools for python.

H.2 Code version linked to outputs

You might successfully implement an automated pipeline and a reproducible environment. However unless you know which version of these was used to produce an output you might well come unstuck!

Make sure that you can track backwards and determine which version of your code produced a particular output.

You can do this by:

- Use git to version control your code
- Make 'atomic' commits which relate to individual Changes
- include the git hash (identifies the code) in the output
  - R: git2r
  - Python: gitpython

Appendix I

DHSC Adopted Style Guides

In line with the easy to read principle DHSC has adopted a single style guide for each language. Please use this style; consistency will make it easier for colleagues to understand, QA and improve your code!

Language	DHSC Adopted Style Guide
R	tidyverse style guide
Python	PEP-8

If you or your team think that the style decision is wrong - please get in touch!

Appendix J

Linters & Code Formatters

Linters are tools that you can use to ensure that you are following a given style guide. The following are recommended:

J.1 R

Styler - http://styler.r-lib.org/

J.2 Python

Black - https://pypi.org/project/black/