DASD Coding Standards

Data Science Hub

Contents

Introduction

This document contains the Data and Analytical Service Directorate's (DASD) coding standards.

0.1 Motivation

The DASD coding standards have been developed to help DASD adopt a unified/consistent approach to writing code, ensuring that our work is high quality, maintainable and reusable and that we are able to collaborate effectively across multiple projects [AMEND].

The purpose of this document is to raise awareness of/hgihlight key coding principles and to briefly explain why we should code in such a way. These coding standards are by no means a list of rules, but seek to provide guidance on how we can achieve 'best practice' in our coding. The more of these you follow, the easier life will be for you and your colleagues. These are open for discussion….

Coding practices should aim to create code that is:

- Reproducible
  - Measures should be put in place to ensure that code can be run across different machines and over time to produce the same results.

- Collaborative
  - Projects should be structured in such a way that we can collaborate effectively across multiple projects
  - Code should be structured efficiently so that others can re-use existing code. This ensures consistency in approach and reduces the need for new code to be written and quality assured.

- Accurate
  - Code should be error free and appropriately quality assured.

- Understandable

- Code should be structured, commented and documented to ensure it can be understood easily by others and passed on to others with minimal additional instruction.

0.2 Guidance structure

The first part of this document contains the four coding principles, with additional details on the practical actions people should take to achieve these.

There is the recommended workflow for coding projects - this can be used to aid understanding of the steps that should be taken in a project and how the coding principles fit into the project lifecycle.

The final section includes a checklist of actions to be taken to achieve the principles. This is a condensed list of the material covered in the coding principles section.

0.3 Contributing

If you think that something is missing or not quite right you should either:
1. Contribute directly
2. Raise an issue

Part I

Principles

Coding Principles

The coding principles have been designed to help DASD adopt a unified/consistent approach to writing code, ensuring that our work is high quality, maintainable and reusable and that we are able to collaborate effectively across multiple projects [AMEND].

Note that the specific actions recommended within this document, although grouped under each of the four principles, do overlap between the principles - they have been placed where they were deemed to fit best.

They are guidance, not The Law - there will always be edge cases, but you should expect to be challenged if you go your own way.

Code should be: [**Update after rewriting individual sections**]

1. **Understandable**

   - Write a README for your project

   - Use sensible defaults unless you have a great reason not to
   - Write functions where needed

   - Stylistic:
     - Apply a linter
     - Ensure variable names are meaningful
     - Code should be correct, clear and concise
     - Handle errors

     - Other team members are users too - treat them with respect

2. **Accurate**

   - Ensure code is reviewed
   - Complete unit testing

3. **Collaborative**

   - Use the github workflow across all projects

- Share the knowledge

4. **Reproducible**

- Manage project dependencies
- Optimise for change

- Analyses should be simple and easy to reproduce on another machine.

Chapter 1

Understandable

It is crucial that we write code in a way that is understandable - through making sure that it is structured, commented and documented appropriately. This will mean that it can be understood easily by others and passed on to other staff with minimal additional instruction. This is clearly aids collboration and accuracy. [Amend]

1.1 Code structure

There should be clear logical separation of parameters, assumptions, data and code, with appropriate documentation.

Data

- MOJ data should never be committed to the Github repository - even a 'private' repo isn't secure. Instead, data should be stored in an s3 bucket or in Athena. [Link to ap guidance].
- Small data files that are not MOJ data can be committed to the Github repo (e.g. parameters, csv containing assumptions, mock data for unit tests, lookup table) - these should be stored in a 'data' folder within the project folder.[Link to github with example project structure]
- Prefer text format (e.g. csv) to Excel
- It should be easy for others to change parameters and data without needed to understand the full codebase.

Documentation

- Your project should include a README - by writing a README for your project you're helping others (and your future self) to be able to understand and run your project. Find our template README here.
- You have added a description to your Github repository and tagged it with appropriate tags This will allow your project to be discoverable and reusable by others.
- Any functions you create should have appropriate documentation
- Your code should be appropriately commented
- Try to make your commit messages as informative as possible

Abstractions

Where appropriate, abstractions should be used if possible, e.g. functions, packages, modules etc. This makes code easier to understand, maintain and extend.

For R, you should use packages as fundamental units of code. For Python, you should factor code out into modules. In javascript, you should generally be using multiple `.js` files. For visualisation code, there should be a separation of concerns between the data model, and the code that visualises your data.

If you end up using a piece of code 3 times, it's probably worth turning it into a function and separating it out into a separate script. For more information on how to write functions, see here. As a general rule, functions should be less than about 50 lines long.

All non trivial functions are documented using the programming language's accepted standard. This means that other users can understand your code more easily. If your functions are short and well documented, there is often little need for additional code comments. For Python follow PEP8 and particularly PEP257. For R, use `roxygen2` to document your functions.

1.2 Coding style

1.2.1 Defaults

There are often many ways of tackling a given problem. As a team, it makes sense to standardise our approach, not because one approach is necessarily better than all others, but because collaboration is easier if there is less diversity in our approaches. This section sets out sensible defaults which you are expected to follow. They are not strict rules, but you will be expected to explain the benefits of alternative approaches if you want to do something different.

Language	Defaults
R	Follow Hadley's Style Guide * 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 `%>%` appropriately, but not always e.g. see here. * Prefer `purrr` to the `apply` family of functions. See here * Use the package name when calling a function. For example, using dplyr::mutate() rather than just mutate() * Use Packrat for R dependencies - it's required by the Analytical Platform if you need to deploy your work. * R Packages are the fundamental unit of reproducable R code.
Python	• Follow [PEP8] • 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
Encoding and CSVs	Use unicode. This means you should convert inputs that include non-ASCII characters to unicode as early as possible in your data processing workflow. If you are outputting to text files, these should be encoded in utf-8. Your output csvs should pass this csv linter.
SQL	Use Postgres or SQLite where possible, rather than other SQL database backends.
GIS	• Use PostGIS as your GIS backend • Prefer conducting your GIS analysis in code, e.g. using SQL, rather than point and click in a GUI • Use QGIS if you need a GUI.

Coding Style

In addition to the above coding defaults, the following actions will help you to write understandable code.

1.2.2 Naming conventions

- Don't be cute or jokey when naming things.
- Names convey meaning - well-named functions & variables can remove the need for a comment and make life a little easier for other readers, including your future self!
- Avoid meaningless names like 'obj' / 'result' / 'foo'.
- Use single-letter variables only where the letter represents a well-known mathematical property (e.g. e = mc^2), or where their meaning is otherwise clear

1.2.3 Clear and concise code

- Choose clarity over cleverness - use advanced language tricks with care.
- Code should be DRY - which stands for 'Don't Repeat Yourself'. - The 'Rule of Three' is a good approach to managing duplication. Less code is usually better - but not at the expense of clarity.

- Ensure that your tests and linters run automatically on all pull requests, if that is possible, by …
- Use code comments judiciously:

  > Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" Improve the code and then document it to make it even clearer." - Steve McConnell

Comments are for explaining why something is needed, not how it works. Appropriate use of comments means it is easier to understand how code works at a later date. Before commenting, consider whether the comment fits better into the function's documentation than in a comment.

1.2.4 Apply a linter

Apply a linter to easily review your code formatting. This means our coding style will be consistent across projects. It will also make it easier for others to understand our code.

In RStudio, the keyboard shortcut 'ctrl+shift+A' will reformat your code and automate some of the process of passing the linter. If you apply the linter as you work, rather than at the end, you will find it much easier to write code that passes the linter first time.

- For R use Lintr and follow Hadley's Style Guide

- For Python, use [pylint] and follow [PEP8]

1.2.5 Helpful error messages

Accept this and code defensively when calling other services. Every HTTP call could error or hang - handle failures appropriately and fail fast. Don't let long-running external calls impact your user experience. Aim to provide useful information to end users and people working on the code, when something fails.

Chapter 2

Accurate

Code should be error free and appropriately quality assured. Alongside simple sense-checking, two of the key mechanisms for ensuring that code is accurate are unit testing and project review.

2.1 Unit testing

Testing our code helps to ensure that it is both correct and robust - unit tests help to give confidence that your code actually does what you think it does. For further guidance on what you should test, see the Coffee and Coding 'Testing' session. [AMEND] - do we need to expand beyond unit testing?

As a broad description, unit tests should exist to check that your actual results match your expected results, in particular, testing any functions you create.

[AMEND] What exactly are we expecting people to test? Unit tests should test, as a minimum, any functions you create. Unit tests exist at the function level, which test a range of parameters. The purpose of these granular tests is to ensure the code continues to give the correct answer in a range of cases, and even in edge cases (where unusual inputs are provided).

[AMEND] * Ensure that your tests and linters run automatically on all pull requests, if that is possible, by setting up TravisCI on your repo. This will probably be more complicated than it is worth if your repo is private due to restrictions that are applied to the free TravisCI accounts

There are a number of tools to enable unit testing.

Testing in R

In R, consider using the testthat package. For an introduction to using testthat, try reading this blog post from Inattentional Coffee or this Towards Data Science

post. For an example unit tests within a project, see here.

Testing in Python

There are various approaches to unit testing in Python. Consider using unittest in the Python core library. Another option is pytest. See also here

Testing in Javascript

See here for testing with javascript

For data vis in Javascript, you need unit tests of routines that manipulate your data or data structures. Visual checks are sufficient of visualisation outputs, but you must make visual checks of the output against real data, and some test datasets that produce predictable output (e.g. where values are set to 1, 0.5 etc.)

2.2 Project Review

Code review provides additional assurance that code logic is correct, and also the review should provide comments on code and problem structuring. For smaller projects, the review only needs to be a simple read-through and sanity check.

Code reviews should be initiated through the creation of a pull request [LINK]. The review should typically involve the reviewer pulling the code to their local machine, testing it, and leaving comments in the pull request.

Remember that it's always easier (for both you and your reviewers) if you commit and push your changes regularly.You should merge branches into master regularly so that reviewers review little and often, rather than attempting to review your entire codebase all at once.

Performing good peer review

When you review someone's pull request you become the gatekeeper to the master branch - this is a *very* important job! If you're tasked with this and you're wondering how to proceed asking yourself these questions is a good place to start…

1: Do I understand what the code is doing? Did it need to be explained to me? Could it be simpler?

2: Are they using packages / libraries sensibly?

3: Does it need to be tested (and is it tested with sufficient coverage)?

4: Does it work? Does it work on my machine?

5: Are there edge cases that might break The Thing?

[AMEND] * Ensure that your tests and linters run automatically on all pull requests, if that is possible, by setting up TravisCI on your repo. This will probably be more complicated than it is worth if your repo is private due to restrictions that are applied to the free TravisCI accounts

If you're reviewing the code of a more experienced coder, this is **a chance to learn** and you have *every* right to ask for an explanation if there's something that is unclear. It's in everyone's interest that you understand what you're reading and it could well be that you don't yet understand it because the author has made a mistake or overcomplicated something. So *don't hold back.*

If you're on the receiving end of feedback, from anyone at all, this is… **a chance to learn!**

Chapter 3

Collaborative

It is vital that we're able to collaborate across projects, to share both workload and expertise, and that this collaboration can occur both simultaneously and over time.

Part of this is managing dependencies so projects can be run on another machine. Another key part is the use of Github for enabling code sharing. Checking the code into Github is a key part of the version control workflow.

3.1 Version Control

Code is version controlled using Git and checked into Github. You can find a guide to using Git with R here You are able to maintain a continuously quality controlled 'master' version of the code, whilst also being able to test new features and functionality.

The 'Master' branch is protected, and contains only QAed code When you create your git project using `git init`, immediately branch off to a development branch e.g. using `git checkout -b dev`, and work on this branch.
Merge code into master using pull requests in github. A protected master branch guarantees that all pull requests have been reviewed before they are merged.

We follow "github flow", to keep branches small and short-lived, and ensure knowledge is shared.

Github-Flow is a working practice that helps to:

1. Maintain overall code quality

2. Facilitate collaboration on a single project

3. Protect the codebase

We have tweaked it a little from what is described on GitHub

There are 6 steps to our process:

1. Create or clone a repo.

```
For example, to clone this repo.
git clone git@github.com:moj-analytical-services/our-coding-standards.git
```

2. Create an issue in Github that describes what you're working on To create an issue, use the Github website.

3. Create a new branch for the work you're about to do, with a name corresponding to the issue

```
To create a new branch and switch onto it.
git checkout -b my-new-sensibly-named-branch
```

4. Make some commits on the new branch.

```
Make some changes then stage each file you've changed - e.g. file1.txt and file2.txt.
git add file1.txt
git add file2.txt
etc

Commit your changes using a descriptive commit message.
git commit

This will take you into your default text editor
Write a descriptive commit message
```

> **Note:** If you have not configured your text editor, you may get stuck in Vim. You can exit using the following command: `:q!`. Then configure your default text editor for Git

```
git config --global core.editor <my-favourite-text-editor>
 Then try again
git commit
```

5. When you're ready, submit a pull request and wait for peer-review.

```
push your branch to the remote repo
git push origin my-new-sensibly-named-branch
then go to github, open a PR and invite at least one reviewer
```

Make sure that you reference the issue in your pull request, by using the hash (#) symbol - see here for further guidance. This makes it easy in future to see what changes were made to the code in response to the issue.

6. To make further changes, just make more commits on the same branch and push them to the remote repo again.
7. Once peer review is complete, and any comments addressed, merge into the master branch using a rebase.
8. The version of master on Github is now ahead of the version of master on your local machine. Bring your local version up to date using `git checkout master`, `git pull`. You are now in sync with Github, and ready to start a new branch.

The **master branch should be 100% functional at all times**, on any machine. Please ensure it is protected and that your tests and / or linters run automatically on all pull requests.

For some further reading we strongly suggest reading this article that explains these git commands and others in a bit more detail.

If you want to test this out, clone this repo and make a contribution :)

Useful links for using github

- A guide to getting started with github
- Github guide on analytical platform guidance
- List of basic git commands

##Share the knowledge {#knowledge}

We also collaborate through sharing knowledge. If you produce something reusable, package it & share it with others.

There are a number of channels through which coding knowledge is shared in DASD:
* Slack channels (key channels are: #data_science, #r, #python)
* Coffee and Coding (resources here)
* DASD training Trello
* R learning resources

Chapter 4

Reproducible

We want to create reproducible code to ensure that it can be used by others, both for collaboration and to allow effective review and accountability, that it keeps working over time (protected from external changes) and so that it can be easily reused by others in their own projects (delete this last one?).

There are a number of steps that we can take to ensure that our code is as reproducible as possible.

4.1 Manage project dependencies

Your project will depend on an number of external factors, such as software or packages. These dependencies may mean that your project won't work on others' machines or may not work on your machine at a later date (e.g. as external packages are updated over time). To ensure that this doesn't become an issue for your project, you should use some kind of dependency management tool.

Dependency management tools

For Python, use Conda. For R, you can use Conda, Packrat or Renv. For javascript, include third party library dependencies in the project as `.js` files

Include a git hash

If practical, the output of your code should include the git hash of the code that produced it. By doing so, the analysis should be more reproducible, there is no ambiguity about the specific code that was used to generate it.

R

You can access the git hash using either of the following code: snippets.

```
library(git2r)
repo <- repository(".")
print(repository_head(repo))
```

or

```
print(system("git rev-parse --short HEAD", intern = TRUE))
```

Python

You can access the git hash using the following code:

```
import subprocess
def get_git_revision_hash():
    return subprocess.check_output(['git', 'rev-parse', 'HEAD'])
def get_git_revision_short_hash():
    return subprocess.check_output(['git', 'rev-parse', '--short', 'HEAD'])
```

4.2 Format

If the output is a report, the write up should be fully reproducible, or as close as possible.

- Avoid workflows that require manually copying and pasting results between documents.
- For Python, consider using Jupyter notebooks. For R, use `rmarkdown`.

4.3 Optimize for change

- Don't try to solve every conceivable problem up-front, instead focus on making your code easy to change when needed.
- Don't prematurely optimize - choose clarity over performance, unless there is a serious performance issue that needs to be addressed.
- Change can come in several forms, including hardware - your code will eventually be run on a colleague's machine or a server somewhere. Without overcomplicating things, write your code with this in mind. For example, use relative paths (e.g. `./file_in_the_project_directory.R` rather than `/Users/my_username/development/my_project/file_in_the_project_directory.R`)

Part II

Resources

Chapter 5

Workflow

The recommended workflow for coding projects in DASD is:

Create/clone github repo
Create issue
Create branch
Set up dependency management
Build in unit testing
Commit changes
Pull request/QA

Chapter 6

Reviewer notes

This document is an early attempt at restructuring the current coding standards. At present, every part of the existing coding standards has been incorporated into this document. This is unlikely to be the case for the final product, but I didn't want to lose any essential information early on.

The minimum coding standards have been incorporated into the document here. The formatting and sizing of the embedded html file isn't ideal at present, but provides an illustration of what could be done. Do we even still need the minimum coding standards? Can we easily put all the requirements in one place rather than splitting into two?

In addition, further information will be added at later stages - particularly in the form of links to external sites that will support the beginner in understanding some of the concepts.

This document hasn't been spellchecked or rigorously tested yet (e.g. that links work etc.). This will be done at a later stage when structure has been agreed.

I've also added a section with the recommended workflow. I wonder if this would ideally be brought up to the front of the beginning of the document to illustrate to beginners how each of the sections of the principles fits into their project work?