DASD Coding Standards

Data Science Hub

Contents

Chapter 1

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?

Chapter 2

Introduction

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

2.1 Motivation

Filler Text
Within DASD we use coding standards to ensure that all the code we produce is:

- Reproducible
- Collaborative
- Accurate
- Understandble

Why do we want these things? And how will adopting these coding standards enable us to acheive these characteristics for our coding?

2.2 Guidance structure

The coding standards are split into two sections:

- Prescriptive minimum coding standards for analytical projects, and
- A set of principles for working on code projects in DASD

The minimum coding standards are required for all projects. The principles have been developed as examples of best practice.

2.3   Contributing

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

Chapter 3

Minimum Coding Standards

3.1 Minimum Standards Checklist

All projects should meet our proportionate minimum coding standards, which can be found here.

This checklist covers:

- Code structure

- Reproducibility

- Development workflow

- Documentation

- Unit testing

- Dependency management

- Packages and versions

3.2 Minimum Standards Table

Minimum AQA Standards for code

This page sets out how to implement existing guidance on AQA for for analytical projects which are written primarily as code (i.e. in a programming langauge). Existing guidance can be found here:

```
\\dom1\data\HQ\102PF\Shared\Group_LCDSHD2\Analytical
Services\AQA - guidance-forms etc\Tailoring Quality
Assurance
```

Existing guidance

Existing guidance provides a framework for categorising your project into nine categories, based on the impact of the analysis and the risks associated with errors.

This guidance sets minimum standards associated with these categories.

When does this guidance apply?

This guidance applies to work that we deliver to our customers. You may perform exploratory data analysis for your own use, and commit this code to Github without adhering to these minimum standards. In this case you should tag the repository as 'exploratory' and make a note of this in your readme. Create a separate repository for the final analysis that will be delivered to customers, which will then be subject to these minimum standards.

Interactive checklist

Select risk

Select scale

Select language(s)

| Low Risk ▾ | | Low Scale ▾ |

Python
R
Javascript

☐ Toggle more details

☐ Toggle 'getting started' advice

Download checklist as marke

Chapter 4

Coding Principles

The following coding principles have been developed to help achieve the following objectives. (I) Ensure we write high quality, maintainable, code; (II) Where possible, ensure our work is reusable; and (III) Collaborate with one another effectively accross multiple projects.

Code should be:

1. **Reproducible**

   - Manage project dependencies
   - Optimise for change

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

2. **Collaborative**

   - Use the github workflow across all projects

   - Share the knowledge

3. **Accurate**

   - Ensure code is reviewed
   - Complete unit testing

4. **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

These ideas are stated in no particular order, and are *always* open to debate. In fact, you are encouraged to contribute.

All development is a trade-off between competing pressures, these principles are meant to help you decide which trade-offs are acceptable.

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.

4.1 Reproducibile

4.1.1 Manage project dependencies

Previously packrat was more widely used, but lots of people seem to use conda now.
* Conda
* Packrat
* Renv

4.1.2 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`)

4.1.3 Format

If the output is a report, the write up is fully reproducible, or as close as possible.
* Avoid workflows that require manually copying and pasting results between documents.
* Your code should be DRY
* For Python, consider using Jupyter notebooks. For R, use `rmarkdown`

4.1.4 Reproducing on another machine

Your project should be tested on someone else's machine There may be software or other dependencies installed on your machine that mean the project doesn't work on others' machines.

Your dependencies are explicitly managed. If your code depends on external packages, these change over time, and need to be enumerated. This guarantees that your code will work on others' machines, and that it will work on your machine in a year's time. For Python, use a Virtual Environment For R, use packrat For javascript, include third party library dependencies in the project as `.js` files

Include a git hash

Some tips on making your analyses simple to reproduce

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.

4.1.4.0.1 R

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

```r
library(git2r)

repo <- repository(".")
print(head(repo))
```

```
## Warning: 'head.git_repository' is deprecated.
## Use 'repository_head' instead.
## See help("Deprecated")

## [8dd5fd] (Local) (HEAD) nikki-rewrite
```

or

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

```
## [1] "8dd5fd0"
```

4.1.4.0.2 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   Collaborative

4.2.1   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.

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

4.2.2   Share the knowledge

If you have knowledge which is unique to you, it is your responsibility to share it. We follow "github flow", to keep branches small and short-lived, and ensure knowledge is shared. You could also present your work at Display DaSH. If you produce something reusable, package it & share it with others. All non-throwaway code should be reviewed - no-one is 100% right, 100% of the time. Be aware that 'throwaway' code has a nasty habit of somehow ending up in completed products.

4.3   Accurate

4.3.1   Review

Ensure that code is reviewed: initiate this through a pull request. Remember that it's always easier (for both you and your reviewers) if you commit and push your changes regularly.

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

All code has been subject to code review. This process has been managed through pull requests, and this evidenced in Github. This should typically involve the reviewer pulling the code to their local machine, testing it, and leaving comments in the pull request. 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. Code review provides additional assurance that code logic is correct, and also the review should provide comments on code and problem structuring.

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?

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!** :)

4.3.2 Unit testing

Use testthat and shinytest for unit testing. The following might be helpful:
* Nice intro to using testthat
* Another intro to testthat
* Example of using testthat

Unit tests exist that test the overall codebase, but not individual functions. Unit tests help give you confidence that your model does what you think it does There are various approaches to unit testing in Python. Consider using unittest in the Python core library. Another option is pytest. See also here In R, you should use the testthat package for unit testing.
You can find examples of how to do unit testing here and here See here for testing with javascript

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 ion a range of cases, and even in edge cases (where unusual inputs are provided). 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.)

4.4 Understandable

4.4.1 Documentation

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.

4.4.2   Use sensible 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.

General

- You should target tidy data structures as part of your work. You should attempt to convert incoming data into tidy format as quickly as possible, and any data that is output that may be used in other projects should be in tidy format.
- If there is a standardised directory structure for your type of project, using it will help people find things. [TODO: Add examples]

You have used the sensible defaults, unless you have a good reason not to and have agreed this with your line manager 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.

R

- 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 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

- 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.

Javascript

- Use Vega or Vega-lite for pre-constructed charting
- Use d3.js for custom charting
- Use leaflet.js for mapping.
- Use Bootstrap as a style template
- Use underscore.js for data manipulation

Finally

- If you think of something else we should be doing or using by default, please clone the repo and submit a pull request featuring that addition :)

4.4.3 Code structure

There should be clear logical separation of parameters, assumptions, data and code.
* 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.

4.4.4   Abstractions

Where appropriate, abstractions should be used if possible, e.g. functions, packages, modules etc. This makes codeeasier 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.

Cut this? If using a notebook (e.g. an R or Jupyter notebook) for your write up, functions are factored out to maintain narrative. Notebooks can also be useful for other purposes, such as functioning as a way of describing exploratory analysis. Where notebooks are used for this type of purpose, they can be more free-form. This makes your logic easier to follow For Python, code should be factored out into modules. For R, factor out code either into a `source()`ed `.R` file or a package

4.4.5   Coding style

For R, follow Hadley's Style Guide For Python, follow [PEP8]

An easy way to check your code formatting is to...

4.4.5.1   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]

4.4.5.2 Package name

It's best practice to use the package name when calling a function. For example, using dplyr::mutate() rather than just mutate().

4.4.5.3 Names have power and purpose - use them wisely

- 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

4.4.5.4 Code should be correct, clear, concise - in that order

- Correct means demonstrably correct - with tests (ideally) and / or documented Quality Assurance. Automated tests are ideal because they allow the code to be refactored with confidence.
- All fixes & new features should include tests to prevent regressions (i.e. reappearance of the bug you just fixed).
- 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.

- To help with clarity and establish patterns within the team, please use either the PEP8 style guide (for Python) or LintR (for R). The relavant linters will help you with this.
- 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.
- 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

Code is appropriately commented. [Comments are for explaining why something is needed, not how it works.] (https://github.com/moj-analytical-services/

our-coding-standards/blob/7e751164d577b521e7f62484a68ee1861f8ae4ac/
they_are_users_too.md#L4) 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.

4.4.5.5 Everything fails at some point

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.

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

If you have to explain how your code works, then your code is not clear enough.
Be nice to your future self (and assume that they will have forgotten the exact
thought process that led to that bit of code you just wrote). Comments are
for explaining why something is needed, not how it works. Make your commit
messages as informative as possible

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