DASD Coding Principles

Data Science Hub

Contents

Introduction

This document contains the Data and Analytical Service Directorate's (DASD) coding principles. These have been developed to help DASD adopt a standard approach for coding projects, ensuring that our work is high quality, maintainable and reusable and that we are able to collaborate effectively.

These coding principles are by no means a list of rules, but seek to provide guidance on how we can achieve 'best practice' in our coding - there will always be edge cases, but you should expect to be challenged if you go your own way. These ideas are always open to debate and you are encouraged to contribute to the discussion.

Document structure

The first part of this document contains the suggested project workflow - this may be helpful in understanding where to apply the principles in the project life-cycle [**AMEND**]

The second part of the documents contains the four coding principles, with details on the practical actions that can be taken to follow the principles.

The final part of the document contains additional resources that may be helpful in understanding how to apply the principles, including a project checklist and a knowledge share resource list.

Contributing

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

Part I

Workflow

Workflow

The following is the recommended workflow for coding projects in DASD. It has been included here to aid understanding of how the coding principles fit into the project life-cycle.

1. **Create/clone Github repo**

- By storing your code in a repo you're making it easier for others to collaborate on your project, to use pieces of your code elsewhere and ensures that your project will be accessible in the future.

2. **Create issue**

- Using issues to communicate what you're working on helps to coordinate tasks between team members and creates a record of completed work, useful for quality assurance purposes.

3. **Create branch**

- Creating a branch means that several people can work on a project simultaneously and protects the main codebase from any unwanted changes.

4. **Set up dependency management**

- Managing your project dependencies means that your project is more likely to be re-runnable in the future and on others' machine.

5. **Build in unit testing**

- Unit tests provide confidence that your code is correct and can be used to pinpoint issues where they occur, making fixes quicker and easier.

6. **Commit changes**

- Committing your changes means that you can recover your work if you make a mistake and provides a record of changes for your reviewer.

7. **Pull request and code review**

- Ensuring that your code is reviewed provides confidence in the accuracy of your code and the opportunity to share knowledge with colleagues.

Part II

Principles

Coding Principles

We want to aim for code that is:

1. **Understandable**

   - Projects should be structured with clear logical separation of parameters, assumptions, data and code, with appropriate documentation.
   - As far as possible, code should follow the style guidance outlined.

2. **Accurate**

   - Projects should include unit testing.
   - All code should be reviewed.

3. **Collaborative**

   - Projects should follow the github workflow.
   - Knowledge should be shared.

4. **Reproducible**

   - Project dependencies should be managed.
   - The format of the output should be chosen with reproducibility in mind.
   - Projects should be optimized for change.

Chapter 1

Understandable

It is crucial that we write code in a way that can be understood easily by others (and our future selves!) and passed on to other staff with minimal additional instruction.

1.1 Code structure

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

Data

- It should be easy for others to change parameters and data without needed to understand the full codebase.
- Where possible, it's generally best practice to use lookup/reference files as much as possible, rather than hard-coding variables. An alternative approach is to store all parameters in one script, making it easy for the variables to be located and updated.
- Small data files that are not MOJ data can be committed to the Github repo (e.g. parameters, csvs containing assumptions, mock data for unit tests, lookup tables) - these should be stored in a 'data' folder within the project folder.[**POPULATE** with link to Github with example project structure]
- Prefer text format (e.g. csv) to Excel

- Please note that unpublished 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. [**POPULATE** with link to ap guidance].

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 should add a description to your Github repository and tag it with appropriate topics. This will allow your project to be discoverable and reusable by others. Github will also automatically tag it according to the language used.

- Any functions you create should have appropriate documentation [**POPULATE** with link to how to do this.] If possible, any code that creates a function should include a clear explanation of the format of both the inputs and outputs. This makes it easier to understand what you can change if you want a different output.

- Try to make your Git commit messages as informative as possible.

- Your code should be appropriately commented. Comments are for explaining why something is needed, not how it works. Note that there are no hard and fast rules on how to do this - this blog) gives an overview on some of the differing opinions on this. [**INSERT EXAMPLE?**]

  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

1.2 Code style

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 more commonality in our approaches.

Defaults

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.

[**AMEND** - check accuracy of these suggestions - are they still in date?]

Language	Defaults
R	• Follow the Tidyverse 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() • R Packages are the fundamental unit of reproducible 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.

Language	Defaults
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.

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

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

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. [**AMEND** - remove? populate links?]
- ~~Ensure that your tests and linters run automatically on all pull requests, if that is possible, by …~~[**POPULATE**]
- Use code comments well (see above)

Linters

A linter is a tool that analyses code to check for programmatic and stylistic errors. You should apply a linter to review your code formatting, which will mean our coding style will be consistent across projects and 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 the Tidyverse Style Guide
- For Python, use [pylint] and follow [PEP8] **POPULATE** links

Error messages

Errors will occur, so code defensively when calling other services. For example, 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. For further guidance on what you should test, see the Coffee and Coding 'Testing' session. [**AMEND** - do we need to expand this section beyond unit testing?]

As a broad description, unit tests should exist to check that your actual results match your expected results.

[**AMEND**] What exactly are we expecting people to test?

Unit tests should test, as a minimum, any functions you create. 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** - this is no longer current] You should 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.

Language	Tools
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.
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
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. 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?(#projdep)

5: Are there any cases that might break it?

6: Is there sufficient documentation?

7: Have they adopted a sensible code style?

AMEND - update and merge with section above * 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 over-complicated 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.

A key tool in enabling collaboration is Github, which can be used for sharing code. Checking the code into Github is part of the version control workflow.

3.1 Version control

Code should be version controlled using Git and checked into Github. You can find a guide to using Git with R here

Within DASD we follow Github-Flow for our projects. Further information and guidance can be found here. [**POPULATE** with a link?]

Github-Flow

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

[**AMEND** - check that this is all up-to-date and correct. Update links where needed.]

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 [**POPULATE** with linnk to automating tests and linters]. A protected master branch guarantees that all pull requests have been reviewed before they are merged.

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 Github links

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

3.2 Cross-team working

POPULATE THIS

3.3 Share the knowledge

We also collaborate through sharing knowledge. If you produce something reusable, please package it and 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 our code to be reproducible so that: it can be used by others (both for collaboration and to allow effective review and accountability); it keeps working over time (protected from external changes); and so that it can be easily reused by others in their own projects (**AMEND** - 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

Language	Tools
R	Use Conda, Packrat or Renv. [**AMEND** - add comment on which is preferable?]
Python	Use Conda
Javascript	Include third party library dependencies in the project as `.js` files

Include a git hash

[**AMEND** - is this still relevant/current?]

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'])
```

Abstractions

Abstractions (e.g. functions, packages, modules etc.) should be used where possible. This makes code easier to understand, maintain and extend.

It will often be the case that will be a pre-existing package or module that contains the functions you need for your project. The best way to find these is usually through a quick google search and, if you're struggling to find what you're looking for, it's worth asking on the relevant slack channel (e.g. #r and #python in the ASD workspace

If you end up using a piece of code three 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 should be documented using the programming language's accepted standard:
* For Python follow PEP8 and particularly PEP257.
* For R, use `roxygen2`. If your functions are short and well documented, there is often little need for additional code comments. [**AMEND** - check links here?]

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 over-complicating 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 III

Resources

Chapter 5

Project Checklist

POPULATE project checklist

Chapter 6

Knowledge Share Resources

POPULATE fully

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
- Online analytical training Trello

PLACEHOLDER - something on cross-departmental resources/code.