DASD Coding Standards

Data Science Hub

Contents

Chapter 1

Introduction

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

1.1 Motivation

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

- Reproducible
- Collaborative
- Accurate
- Understandble

Break it down by each section. High quality: how do adopting CS make the code high quality? Understandable: how do adopting CS make the code understandable? Reproducible: how do adopting CS make the code repdocuible?

1.2 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.

1.3 Further guidance

#1: Reviewing a pull request

#2: Manual quality assurance

#3: Writing an amazing README.md for your project

1.4 Contributing

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

Chapter 2

Minimum Coding Standards

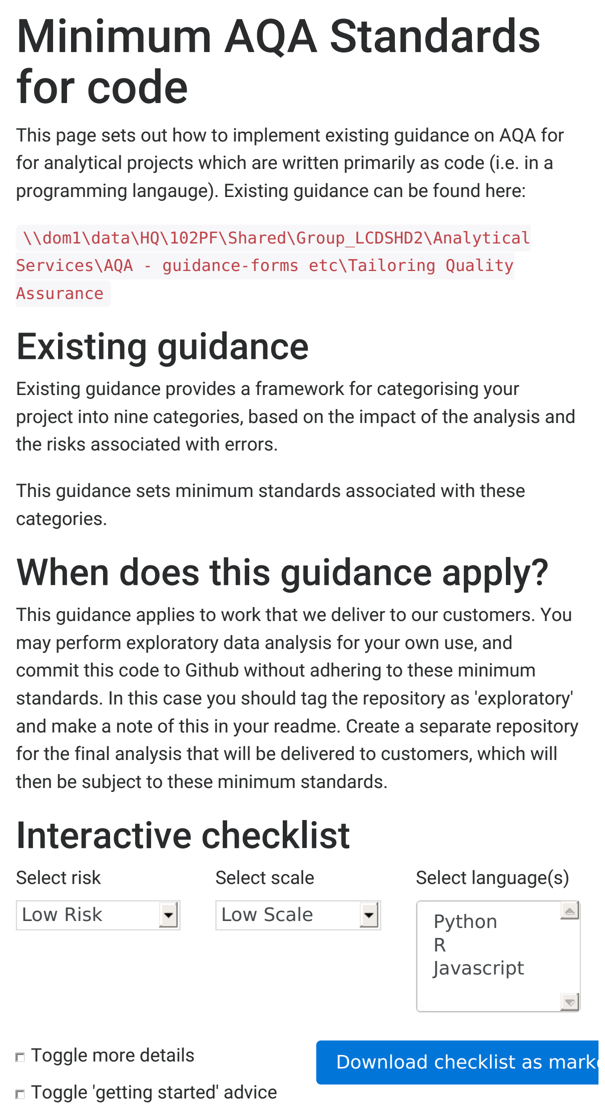
2.1 Minimum Standards Checklist

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

2.2 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: Low Risk

Select scale: Low Scale

Select language(s): Python, R, Javascript

Chapter 3

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**
   - 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.

3.1 Reproducibile

3.1.1 Manage project dependencies

3.1.2 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`)

3.1.3 Reproducing on another machine

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.

3.1.3.0.1 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")
```

```
## [be1429] (Local) (HEAD) nikki-restructure
```

or

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

```
## [1] "be14290"
```

3.1.3.0.2 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'])"'

3.2 Collaborative

3.2.1 Version Control

Github-Flow is a working practice that helps to:

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 :)

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.

3.3 Accurate

3.3.1 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.

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!** :)

3.3.2 Unit testing

Use testthat and shinytest for unit testing. LINKS.

3.4 Understandable

3.4.1 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]

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 :)

3.4.2 Separate functions out

3.4.3 Coding style

Apply a linter

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

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

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.

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