Netherlands eScience Center

Software Development Guide

netherlands eScience center

Table of Contents

DOI 10.5281/zenodo.4020564

Link Checker failing

Guide

This is a guide to software development at the Netherlands eScience Center. It both serves as a source of information for how we work at the eScience Center, and as a basis for discussions and reaching consensus on this topic.

Read The Turing Way instead

If you are looking for an overall picture of best practices, read The Turing Way first. We joined forces with that guide for most of our generic research software engineering advice. Because The Turing Way is programming language agnostic, this guide provides addtional specific **language guides**. We also provide **technology guides** on digital technologies we use often in our projects with research partners. *Please be aware that most remaining **best practices** content is unmaintained, be careful when using it.* We plan on removing that content (#286).

If you would like to contribute to this book see CONTRIBUTING.md.

Page maintainer: Patrick Bos @egpbos

This chapter provides practical info on each of the main programming languages of the Netherlands eScience Center.

This info is (on purpose) high level, try to provide "default" options, and mostly link to more info.

Each chapter should contain:

- Intro: philosophy, typical usecases.
- Recommended sources of information
- Installing compilers and runtimes
- Editors and IDEs
- Coding style conventions
- Building and packaging code
- Testing
- Code quality analysis tools and services
- Debugging and Profiling
- Logging
- Writing documentation
- Recommended additional packages and libraries
- Available templates

Preferred Languages

At the Netherlands eScience Center we prefer Java and Python over C++ and Perl, as these languages in general produce more sustainable code. It is not always possible to choose which libraries we use, as almost all projects have existing code as a starting point.

(In alphabetical order)

- Java
- JavaScript (preferably Typescript)
- Python
- OpenCL and CUDA
- R

Selecting tools and libraries

On GitHub there is a concept of an "awesome list", that collects awesome libraries and tools on some topic. For instance, here is one for Python: https://github.com/vinta/awesome-python

Now, someone has been smart enough to see the pattern, and has created an awesome list of awesome lists: https://awesome.re/

Highly recommented to get some inspiration on available tools and libraries!

Development Services

To do development in any language you first need infrastructure (code hosting, ci, etc). Luckily a lot is available for free now.

See this list: https://github.com/ripienaar/free-for-dev

Bash

Page maintainer: Bouwe Andela @bouweandela

Bash is both a command line interface, also known as a **shell**, and a scripting language. On most Linux distributions, the Bash shell is the default way of interacting with the system. Zsh is an alternative shell that also understands the Bash scripting language, this is the default shell on recent versions of Mac OS. Both Bash and Zsh are available for most operating systems.

At the Netherlands eScience Center, Bash is the recommended shell scripting language because it is the most commonly used shell language and therefore the most convenient for collaboration. To facilitate mutual understanding, it is also recommended that you are aware of the shell that your collaborators are using and that you write documentation with this in mind. Using the same shell as your collaborators is a simple way of making sure you are always on the same page.

In this chapter, a short introduction and best practices for both interactive and use in scripts will be given. An excellent tutorial introducing Bash can be found here. If you have not used Bash or another shell before, it is recommended that you follow the tutorial before continuing reading. Learning to use Bash is highly recommended, because after some initial learning, you will be more efficient and have a better understanding of what is going on than when clicking buttons from the graphical user interface of your operating system or integrated development environment.

Interactive use

If you are a (research) software engineer, it is highly recommended that you learn

- the keyboard shortcuts
- how to configure Bash aliases
- the name and function of commonly used command line tools

Bash keyboard shortcuts

An introduction to bash keyboard shortcuts can be found here. Note that Bash can also be configured such that it uses the *vi* keyboard shortcuts instead of the default *emacs* ones, which can be useful if you prefer vi.

Bash aliases

Bash aliases allow you to define shorthands for commands you use often. Typically these are defined in the `~/.bashrc` or `~/.bash_aliases` file.

Commonly used command line tools

It is recommended that you know at least the names and use of the following command line tools. The details of how to use a tool exactly can easily be found by searching the internet or using `man` to read the manual, but you will be vastly more efficient if you already know the name of the command you are looking for.

Working with files

- `ls` - List files and directories
- `tree` - Graphical representation of a directory structure
- `cd` - Change working directory
- `pwd` - Show current working directory
- `cp` - Copy a file or directory
- `mv` - Move a file or directory
- `rm` - Remove a file or directory
- `mkdir` - Make a new directory
- `touch` - Make a new empty file or update its access and modification time to the current time
- `chmod` - Change the permissions on a file or directory
- `chown` - Change the owner of a file or directory
- `find` - Search for files and directories on the file system
- `locate` , `updatedb` - Search for files and directories quickly using a database
- `tar` - (Un)pack .tar or .tar.gz files
- `unzip` - Unpack .zip files
- `df` , `du` - Show free space on disk, show disk space usage of files/folders

Working with text

Here we list the most commonly used Bash tools that are built to manipulate *lines of text*. The nice thing about these tools is that you can combine them by streaming the output of one tool to become the input of the next tool. Have a look at the tutorial for an introduction. This can be done by creating pipelines with the pipe operator `|` and by redirecting text to output streams or files using redirection operators like `>` for output and `<` for input to a command from a text file.

- `echo` - Repeat some text
- `diff` - Show the difference between two text files
- `grep` - Search for lines of text matching a simple string or regular expressions

- `sed` - Edit lines of text using regular expressions
- `cut` - Select columns from text
- `cat` - Print the content of a file
- `head` - Print the first n lines
- `tail` - Print the last n lines
- `tee` - Read from standard input and write to standard output and file
- `less` - Read text
- `sort` - Sort lines of text
- `uniq` - Keep unique lines
- `wc` - Count words/lines
- `nano` , `emacs` , `vi` - Interactive text editors found on most Unix systems

Working with programs

- `man` - Read the manual
- `ps` - Print all currently running programs
- `top` - Interactively display all currently running programs
- `kill` - Stop a running program
- `\time` - Collect statistics about resource usage such as runtime, memory use, storage access (the `\` in front is needed to run the `time` program instead of the bash builtin function with the same name)
- `which` - Find which file will be executed when you run a command
- `xargs` - Run programs with arguments in parallel

Working with remote systems

- `ssh` - Connect to a shell on a remote computer
- `rsync` - Copy files between computers using SSH/SFTP
- `lftp` - Copy files between computers using FTP
- `wget` , `curl` - Copy a file using https or make a request to a remote API
- `scp` , `sftp` , `ftp` - Simple tools for transferring files over (S)FTP - not recommended
- `who` - show who is logged on
- `screen` - Run multiple bash sessions and keep them running even when you log out

Installing software

- `apt` - The default package manager on Debian based Linux distributions
- `yum` , `dnf` - The default package manager on RedHat/Fedora based Linux distributions
- `brew` - A package manager for MacOS
- `conda` - A package manager that supports many operating systems
- `pip` - The Python package manager

- `docker` , `singularity` - Run an entire Linux operating system including software from a container

Miscellaneous

- `bash` , `zsh` - The command to start Bash/Zsh
- `history` - View all past commands
- `fg` , `bg` - Move a program to the foreground, background, useful with Ctrl+Z
- `su` - Switch user
- `sudo` - Run a command with root permissions

For further inspiration, see this extensive list of command line tools.

Scripts

It is possible to write bash scripts. This is done by writing the commands that you would normally use on the command line in text file and e.g. running the file with `bash some-file.sh` .

However, doing this is only recommended if there really are no other options. If you have the option to write a Python script instead, that is the recommended way to go. This will bring you all the advantages of a fully-fledged programming language (such as libraries, frameworks for testing and documentation) and Python is the recommended programming language at the Netherlands eScience Center. If you do not mind having an extra dependency and would like to use the features and commands available in the shell from Python, the sh library is a nice option.

Disclaimer: if you are an experienced Bash developer, there might be situations where using a Bash script solves your problem faster or in a more portable way than a Python script. Do take take a moment to think about whether such a solution is easy to contribute to for collaborators and will be easy to maintain in the future, as the number of features, supported systems, and code paths grows.

When writing a bash script, always use `shellcheck` to make sure that your bash script is as likely to do what you think it should do as possible.

In addition to that, always start the script with

```bash
set -euo pipefail
```

this will stop the script if there is

- `-e` a command that exits with a non-zero exit code

- `-o pipefail` a command in a pipe that exits with a non-zero exit code
- `-u` an undefined variable in your script

an exit code other than zero usually indicates that an error occurred. If needed, you can temporarily allow this kind of error for a single line by wrapping it like this

```bash
set +e
false  # A command that returns a non-zero exit code
set -e
```

Further resources

- Bash Tutorial
- Bash Cheat sheet
- The Bash Reference Manual or use `man bash`
- Oh My Zsh offers an extensive set of themes and shortcuts for the Zsh

Getting started

Page maintainer: Ewan Cahen @ewan-escience

To learn about JavaScript, view the presentations by Douglas Crockford:

- Crockford on JavaScript
- JavaScript: The Good Parts
- JavaScript trilogy:
    - The JavaScript Programming Language (1h50m)
    - Theory of the DOM (1h18m)
    - Advanced JavaScript (1h07m)

In this video (47m04s), Nicholas Zakas talks about sustainability aspects, such as how to write maintainable JavaScript, how to do JavaScript testing, and good programming style (much needed in JavaScript). Among others, he mentions the following style guides:

- Google's style guide for JavaScript;
- Crockford's style guide integrates with JSLint, which in turn is available as a plugin for Eclipse.
- Zakas has also written an excellent book on writing maintainable JavaScript, also within the context of working in teams. The appendix contains a style guide with explanation.

These video tutorials (totaling a couple of hours) are useful if you're just starting with learning the JavaScript language.

Another source of information for JavaScript is the MDN Web Docs

https://developer.mozilla.org/en-US/docs/Learn

Additionally, see the W3Schools page about JavaScript best practices

Frameworks

Before you pick a framework, you should first consider what you are trying to build.

- If you're building a (more traditional) website with mostly static content, like an info page for an event or a blog, whose content doesn't adapt to the visitor, consider using a static site generator like Jekyll or Hugo or Docusaurus for writing documentation. An advantage of this is that static sites can be hosted on GitHub for free, which uses Jekyll by default (but you can use other static site generators as well).
- If you're building a website that is not very interactive, but that many people have to edit, and when a static site generator is too technical, consider using WordPress. Many hosting providers support WordPress out of the box.

- When you need light interactivity, the options above can be combined with libraries like jQuery, Alpine.js, htmx or you can write the JavaScript yourself.
- When you want to build a website that has high interactivity with its users, something you would call an "application" rather than a "website", consider using htmx or one of the JavaScript frameworks below.

Currently, the most popular frameworks are (ordered by popularity according to the StackOverflow 2024 Developer Survey)

- React
- Angular
- Vue.js
- Svelte
- SolidJS

A good video summary on the history of these frameworks and the problems they try to solve can be found here.

React

React is a framework which can used to create interactive User Interfaces by combining components. It is developed by Facebook. It is by far the most popular framework, resulting in a huge choice of libraries and a lot of available documentation. Contrary to most other frameworks, React apps are typically written in JSX instead of plain HTML, CSS and JS.

Where other frameworks like Angular and Vue.js include rendering, routing and, state management functionality, React only does rendering, so other libraries must be used for routing and state management. Redux can be used to let state changes flow through React components. React Router can be used to navigate the application using URLs. Or you can use a so-called "meta-framework" like Next.js.

To create a React application, the official documentation recommends to start with a meta-framework. Alternatively, you can use the tool Create React App, optionally with TypeScript.

Angular

Angular is a application framework by Google written in TypeScript. It is a full-blown framework, with many features included. It is therefore more used in enterprises and probably overkill for your average scientific project. Read more about what Angular is in the documentation.

To create a Angular application see the installation docs.

Angular also has a meta-framework called Analog.

Vue.js

Vue.js is an open-source JavaScript framework for building user interfaces. Read about the use cases for Vue and reasons to use it in their introduction.

To create a Vue application, read the quick start. It also has info on using TypeScript with Vue.

A meta-framework for Vue is Nuxt.

Svelte

Svelte is a UI framework, that differs with most other frameworks in that is uses a compiler before shipping JavaScript to the client. Svelte applications are written in HTML, CSS and JS. Read more about Svelte in their overview.

In their documentation, they recommend to use their meta-framework SvelteKit to create a Svelte application. It also supports TypeScript.

Solid.js

A UI framework that focuses on performance and being developer friendly. Like React, it uses JSX. Read more about Solid here.

To create a Solid application, check out the quick start. They also support TypeScript.

Solid has a meta-framework called SolidStart.

JavaScript outside of the browser

Most JavaScript is run in web browsers, but if you want to run it outside of a browser (e.g. as a server or to run a script locally), you'll need a JavaScript **runtime**. These are the main runtimes available:

- Node.js is the most used runtime, mainly for being the only available runtime for a long time. This gives the advantage that there is a lot of documentation available (official and unofficial, e.g. forums) and that many tools are available for Node.js. It comes with a package manager (npm) that allows you to install packages from a huge library. Its installation instructions can be found here.
- Deno can be seen as a successor to Node.js and tries to improve on it in a few ways, most notably:
    - built-in support for TypeScript

    - a better security model
    - built-in tooling, like a linter and formatter
    - compiling to standalone executables

  Its installation instructions can be found here

- Bun, the youngest runtime of the three. Its focus is on speed, reduced complexity and enhanced developer productivity (read more here). Just like Deno, it comes with built-in TypeScript support, can compile to standalone executables and it aims to be fully compatible with Node.js. Its installation instructions can be found here.

A more comprehensive comparison can be found in this guide.

Which runtime to choose?

To answer this question, you should consider what is important for you and your project.

Choose Node.js if:

- you need a stable, mature and a well established runtime with a large community around it;
- you need to use dependencies that should most likely "just work";
- you cannot convince the people you work with to install something else;
- you don't need any particular feature of any of its competitors.

Choose Deno if:

- you want a relatively mature runtime with a lot of features built in;
- you want out-of-the-box TypeScript support;
- you like its security model;
- you want a complete package with a linter and formatter included;
- you don't mind spending some time if something does not work directly.

Choose Bun if:

- you are willing to take a risk using a relatively new runtime;
- you want out-of-the-box TypeScript support;
- you want to use one of Bun's particular features;
- you need maximum performance (though you should benchmark for your use case first and consider using a different programming language).

Editors and IDEs

These are some good JavaScript editors:

- WebStorm by JetBrains. It is free (as in monetary cost) for non-commercial use; otherwise you have to buy a licence. Most of its features are also available in other IDEs of JetBrains, like IntelliJ IDEA ultimate, PyCharm professional and Rider. You can compare the products of JetBrains here. Note that the free version of WebStorm will collect data anonymously, without the option to disable it. WebStorm comes with a lot of functionality included, but also gives access to a Marketplace of plugins.
- Visual Studio Code, an open source and free (as in monetary cost) editor by Microsoft. By default, it collects telemetry data, but that can be disabled. VSCode has a limited feature set out of the box, which can be enhanced with extensions.

Debugging

In web development, debugging is typically done in the browser.

- The best debugging tool suite is currently the debugger built into the Google Chrome web browser, and its open-source counterpart, Chromium. It can watch variables, step through the code, lets you monitor network traffic, and much more. Activate the debugger through the F12 key.
- On Firefox, use either the built-in debugging functionality (again accessible through the F12 button) or install the Firefox Developer Tools.
- Microsoft has a debugging toolset called 'F12' for their Internet Explorer and Edge browsers. It offers similar capability as that of Google Chrome, Chromium, and Firefox.
- In Safari on OS X, press ⊠⊠U.

Sometimes the JavaScript code in the browser is not an exact copy of the code you see in your development environment, for example because the original source code is minified/uglified or transpiled before it's loaded in the browser. All major browsers can now deal with this through so-called *source maps*, which instruct the browser which symbol/line in a javascript file corresponds to which line in the human-readable source code. Look for the 'create sourcemaps' option when using minification/uglification/transpiling tools.

Hosting data files

To load data files with JavaScript you can't use any file system URLs due to safety restrictions. You should use a web server (which may still serve files that are local). A simple webserver can be started from the directory you want to host files with:

```bash
python3 -m http.server 8000
```

Then open the web browser to http://localhost:8000.

Documentation

JSDoc works similarly to JavaDoc, in that it parses your JavaScript files and automatically generates HTML documentation.

Testing

- Jasmine, a behavior-driven development framework for testing JavaScript code.
- Karma, Test runner, runs tests in web browser with code coverage. Use PhantomJS as headless web browser on CI-servers.
- Tape, a minimal testing framework that helps remove some of the black-box approach of some of the other frameworks.
- Jest, a test framework from Facebook which is integrated into the Create React App

Web based tests

To interact with web-browsers use Selenium.

Test with

- Local web browser
- Web browsers hosted by Sauce Labs, it has a matrix of web-browsers and Operating Systems. Free for open source projects.

Coding style

See general front dev guidelines and Airbnb JavaScript Style Guide.

Use a linter like eslint to detect errors and potential problems.

Showing code examples

You can use jsfiddle, which shows you a live preview of your web page while you fiddle with the underlying HTML, JavaScript and CSS code.

Code quality analysis tools and services

- Code climate can analyze JavaScript (and Ruby, PHP). For example project see https://codeclimate.com/github/NLeSC/PattyVis

- Codacy can analyze many different languages using open source tools.
- SonarCloud is an open platform to manage code quality which can also show code coverage and count test results over time. Can analyze Java (best supported), C, C++, Python, JavaScript and TypeScript. For example project see https://sonarcloud.io/dashboard?id=e3dchem%3Amolviewer

TypeScript

https://www.typescriptlang.org/

TypeScript is a typed superset of JavaScript which compiles to plain JavaScript. TypeScript adds static typing to JavaScript, which makes it easier to scale up in people and lines of code.

At the Netherlands eScience Center we prefer TypeScript to JavaScript as it will lead to more sustainable software.

This section highlights the differences with JavaScript. For topics that seem to be missing, like IDEs, code style etc., see the respective JavaScript section.

Getting Started

To learn about TypeScript, the following resources are available:

- Official TypeScript documentation and tutorial
- Single video tutorial and playlist tutorial
- Tutorials on debugging TypeScript in Chrome and Firefox. If you are using a framework, consult the documentation of that framework for additional ways of debugging
- The Definitive TypeScript 5.0 Guide
- The W3Schools TypeScript tutorial

Quickstart

To install TypeScript compiler run:

```shell
npm install -g typescript
```

Dealing with Types

In TypeScript, variables are typed and these types are checked. This implies that when using libraries, the types of these libraries need to be installed. More and more libraries ship with type declarations in them so they can be used directly. These libraries will have a "typings" key in their package.json. When a library does not ship with type declarations then the libraries `@types/<library-name>` package must be installed using npm:

```shell
npm install --save-dev @types/<library-name>
```

For example say we want to use the `react` package which we installed using `npm` :

```shell
npm install react --save
```

To be able to use its functionality in TypeScript we need to install the typings.

Install it with:

```shell
npm install --save-dev @types/react
```

The `--save-dev` flag saves this installation to the package.json file as a development dependency. Do not use `--save` for types because a production build will have been transpiled to JavaScript and has no use for TypeScript types.

Debugging

In web development, debugging is typically done in the browser. TypeScript cannot be run directly in the web browser, so it must be transpiled to JavaScript. To map a breakpoint in the browser to a line in the original TypeScript file source maps are required. Most frameworks have a project build system which generate source maps.

Documentation

It seems that TypeDoc is a good tool to use.

Python

Page maintainer: Bouwe Andela @bouweandela

Python is the "dynamic language of choice" of the Netherlands eScience Center. We use it for data analysis and data science projects using the SciPy stack and Jupyter notebooks, and for many other types of projects: workflow management, visualization, NLP, web-based tools and much more. It is a good default choice for many kinds of projects due to its generic nature, its large and broad ecosystem of third-party modules and its compact syntax which allows for rapid prototyping. It is not the language of maximum performance, although in many cases performance critical components can be easily replaced by modules written in faster, compiled languages like C(++) or Cython.

The philosophy of Python is summarized in the Zen of Python. In Python, this text can be retrieved with the `import this` command.

Project setup

When starting a new Python project, consider using our Python template. This template provides a basic project structure, so you can spend less time setting up and configuring your new Python packages, and comply with the software guide right from the start.

Use Python 3, avoid 2

Python 2 and Python 3 have co-existed for a long time, but starting from 2020, development of Python 2 is officially abandoned, meaning Python 2 will no longer be improved, even in case of security issues. If you are creating a new package, use Python 3. It is possible to write Python that is both Python 2 and Python 3 compatible (e.g. using Six), but only do this when you are 100% sure that your package won't be used otherwise. If you need Python 2 because of old, incompatible Python 2 libraries, strongly consider upgrading those libraries to Python 3 or replacing them altogether. Building and/or using Python 2 is probably discouraged even more than, say, using Fortran 77, since at least Fortran 77 compilers are still being maintained.

- Things you’re probably not using in Python 3 – but should
- Six: Python 2 and 3 Compatibility Library
- 2to3: Automated Python 2 to 3 code translation
- python-modernize: wrapper around 2to3

Learning Python

- A popular way to learn Python is by doing it the hard way at http://learnpythonthehardway.org/
- Using `pylint` and `yapf` while learning Python is an easy way to get familiar with best practices and commonly used coding styles

Dependencies and package management

To install Python packages use `pip` or `conda` (or both, see also what is the difference between pip and conda?).

If you are planning on distributing your code at a later stage, be aware that your choice of package management may affect your packaging process. See Building and packaging for more info.

Use virtual environments

We strongly recommend creating isolated "virtual environments" for each Python project. These can be created with `venv` or with `conda`. Advantages over installing packages system-wide or in a single user folder:

- Installs Python modules when you are not root.
- Contains all Python dependencies so the environment keeps working after an upgrade.
- Keeps environments clean for each project, so you don't get more than you need (and can easily reproduce that minimal working situation).
- Lets you select the Python version per environment, so you can test code compatibility between Python versions

Pip + a virtual environment

If you don't want to use `conda`, create isolated Python environments with the standard library `venv` module. If you are still using Python 2, `virtualenv` and `virtualenvwrapper` can be used instead.

With `venv` and `virtualenv`, `pip` is used to install all dependencies. An increasing number of packages are using `wheel`, so `pip` downloads and installs them as binaries. This means they have no build dependencies and are much faster to install.

If the installation of a package fails because of its non-Python extensions or system library dependencies and you are not root, you could switch to `conda` (see below).

Conda

Conda can be used instead of venv and pip, since it is both an environment manager and a package manager. It easily installs binary dependencies, like Python itself or system libraries. Installation of packages that are not using `wheel` , but have a lot of non-Python code, is much faster with Conda than with `pip` because Conda does not compile the package, it only downloads compiled packages. The disadvantage of Conda is that the package needs to have a Conda build recipe. Many Conda build recipes already exist, but they are less common than the `setuptools` configuration that generally all Python packages have.

There are two main "official" distributions of Conda: Anaconda and Miniconda (and variants of the latter like miniforge, explained below). Anaconda is large and contains a lot of common packages, like numpy and matplotlib, whereas Miniconda is very lightweight and only contains Python. If you need more, the `conda` command acts as a package manager for Python packages. If installation with the `conda` command is too slow for your purposes, it is recommended that you use `mamba` instead.

For environments where you do not have admin rights (e.g. DAS-6) either Anaconda or Miniconda is highly recommended since the installation is very straightforward. The installation of packages through Conda is very robust.

A possible downside of Anaconda is the fact that this is offered by a commercial supplier, but we don't foresee any vendor lock-in issues, because all packages are open source and can still be obtained elsewhere. Do note that since 2020, Anaconda has started to ask money from large institutes for downloading packages from their main channel (called the `default` channel) through `conda` . This does not apply to universities and most research institutes, but could apply to some government institutes that also perform research and definitely applies to large for-profit companies. Be aware of this when choosing the distribution channel for your package. An alternative, community-driven Conda distribution that avoids this problem altogether because it only installs packages from `conda-forge` by default is miniforge. Miniforge includes both the faster `mamba` as well as the traditional `conda` .

Building and packaging code

Making an installable package

To create an installable Python package you will have to create a `pyproject.toml` file. This will contain three kinds of information: metadata about your project, information on how to build and install your package, and configuration settings for any tools your project may use. Our Python template already does this for you.

Project metadata

Your project metadata will be under the `[project]` header, and includes such information as the name, version number, description and dependencies. The Python Packaging User Guide has more information on what else can or should be added here. For your dependencies, you should keep version constraints to a minimum; use, in order of descending preference: no constraints, lower bounds, lower + upper bounds, exact versions. Use of `requirements.txt` is discouraged, unless necessary for something specific, see the discussion here.

It is best to keep track of direct dependencies for your project from the start and list these in your `pyproject.toml` If instead you are writing a new `pyproject.toml` for an existing project, a recommended way to find all direct dependencies is by running your code in a clean environment (probably by running your test suite) and installing one by one the dependencies that are missing, as reported by the ensuing errors. It is possible to find the full list of currently installed packages with `pip freeze` or `conda list`, but note that this is not ideal for listing dependencies in `pyproject.toml`, because it also lists all dependencies of the dependencies that you use.

Build system

Besides specifying your project's own metadata, you also have to specify a build-system under the `[build-system]` header. We currently recommend using `hatchling` or `setuptools`. Note that Python's build system landscape is still in flux, so be sure to look upthe some current practices in the packaging guide's section on build backends and authoritative blogs like this one. One important thing to note is that use of `setup.py` and `setup.cfg` has been officially deprecated and we should migrate away from that.

Tool configuration

Finally, `pyproject.toml` can be used to specify the configuration for any other tools like `pytest`, `ruff` and `mypy` your project may use. Each of these gets their own section in your `pyproject.toml` instead of using their own file, saving you from having dozens of such files in your project.

Installation

When the `pyproject.toml` is written, your package can be installed with

```
pip install -e .
```

The `-e` flag will install your package in editable mode, i.e. it will create a symlink to your package in the installation location instead of copying the package. This is convenient when developing, because any changes you make to the source code will immediately be available for use in the installed version.

Set up continuous integration to test your installation setup. You can use `pyroma` as a linter for your installation configuration.

Packaging and distributing your package

For packaging your code, you can either use `pip` or `conda`. Neither of them is better than the other -- they are different; use the one which is more suitable for your project. `pip` may be more suitable for distributing pure python packages, and it provides some support for binary dependencies using **`wheels`**. `conda` may be more suitable when you have external dependencies which cannot be packaged in a wheel.

Build via thePython Package Index (PyPI) so that the package can be installed with pip

- General instructions
- We recommend to configure GitHub Actions to upload the package to PyPI automatically for each release.
  - For new repositories, it is recommended to use trusted publishing because it is more secure than using secret tokens from GitHub.
    - For a workflow using secret tokens instead, see this example workflow in DIANNA.
  - You can follow these instructions to set up GitHub Actions workflows with trusted publishing.
    - The **`verbose`** option for pypi workflows is useful to see why a workflow failed.
    - To avoid unnecessary workflow runs, you can follow the example in the sirup package: manually trigger pushes to pypi and investigate potential bugs during this process with a manual upload.
- Manual uploads with twine
  - Because PyPI and Test PyPI require Two-Factor Authentication per January 2024, you need to mimick GitHub's trusted publishing to publish manually with `twine`.
  - You can follow the section on "The manual way" as described here.
- Additional guidelines:
  - Packages should be uploaded to PyPI using your own account
  - For packages developed in a team or organization, it is recommended that you create a team or organizational account on PyPI and add that as a collaborator with the owner rule. This will allow your team or organization to maintain the package even if individual contributors at some point move on to do other things. At the Netherlands eScience Center, we are a fairly small organization, so we use a single backup account (`nlesc`).
  - When distributing code through PyPI, non-python files (such as `requirements.txt`) will not be packaged automatically, you need to add them to a `MANIFEST.in` file.
  - To test whether your distribution will work correctly before uploading to PyPI, you can run `python -m build` in the root of your repository. Then try installing your package with `pip install dist/<your_package>tar.gz.`

  - `python -m build` will also build Python wheels, the current standard for distributing Python packages. This will work out of the box for pure Python code, without C extensions. If C extensions are used, each OS needs to have its own wheel. The manylinux Docker images can be used for building wheels compatible with multiple Linux distributions. Wheel building can be automated using GitHub Actions or another CI solution, where you can build on all three major platforms using a build matrix.

Build using conda

- **Make use of conda-forge whenever possible**, since it provides many automated build services that save you tons of work, compared to using your own conda repository. It also has a very active community for when you need help.
- Use BioConda or custom channels (hosted on GitHub) as alternatives if need be.

Editors and IDEs

Every major text editor supports Python, either natively or through plugins. At the Netherlands eScience Center, some popular editors or IDEs are:

- vscode holds the middle ground between a lightweight text editor and a full-fledged language-dedicated IDE.
- vim or `emacs` (don't forget to install plugins to get the most out of these two), two versatile classic powertools that can also be used through remote SSH connection when needed.
- JetBrains PyCharm is the Python-specific IDE of choice. PyCharm Community Edition is free and open source; the source code is available in the python folder of the IntelliJ repository.

Coding style conventions

The style guide for Python code is PEP8 and for docstrings it is PEP257. We highly recommend following these conventions, as they are widely agreed upon to improve readability. To make following them significantly easier, we recommend using a linter.

Many linters exists for Python. The most popular one is currently Ruff. Although it is new (see the website for the complete function parity comparison with alternatives), it works well and has an active community. An alternative is `prospector` , a tool for running a suite of linters, including, among others pycodestyle, pydocstyle, pyflakes, pylint, mccabe and pyroma. Some of these tools have seen decreasing community support recently, but it is still a good alternative, having been a defining community default for years.

Most of the above tools can be integrated in text editors and IDEs for convenience.

Autoformatting tools like `yapf` and `black` can automatically format code for optimal readability. `yapf` is configurable to suit your (team's) preferences, whereas `black` enforces the style chosen by the `black` authors. The `isort` package automatically formats and groups all imports in a standard, readable way.

Ruff can do autoformatting as well and can function as a drop-in replacement of `black` and `isort`.

Testing

Use pytest as the basis for your testing setup. This is preferred over the `unittest` standard library, because it has a much more concise syntax and supports many useful features.

It has many plugins. For linting, we have found `pytest-pycodestyle`, `pytest-pydocstyle`, `pytest-mypy` and `pytest-flake8` to be useful. Other plugins we had good experience with are `pytest-cov`, `pytest-html`, `pytest-xdist` and `pytest-nbmake`.

Creating mocks can also be done within the pytest framework by using the `mocker` fixture provided by the `pytest-mock` plugin or by using `MagicMock` and `patch` from `unittest`. For a general explanation about mocking, see the standard library docs on mocking.

To run your test suite, it can be convenient to use `tox`. Testing with `tox` allows for keeping the testing environment separate from your development environment. The development environment will typically accumulate (old) packages during development that interfere with testing; this problem is avoided by testing with `tox`.

Code coverage

When you have tests it is also a good to see which source code is exercised by the test suite. Code coverage can be measured with the coverage Python package. The coverage package can also generate html reports which show which line was covered. Most test runners have have the coverage package integrated.

The code coverage reports can be published online using a code quality service or code coverage services. Preferred is to use one of the code quality service which also handles code coverage listed below. If this is not possible or does not fit then use a generic code coverage service such as Codecov or Coveralls.

Code quality analysis tools and services

Code quality service is explained in the The Turing Way. There are multiple code quality services available for Python, all of which have their pros and cons. See The Turing Way for links to lists of possible services. We currently setup Sonarcloud by default in our Python template. To reproduce the Sonarcloud pipeline locally, you can use SonarLint in your IDE. If you use another editor, perhaps it is more convenient to pick another service like Codacy or Codecov.

Debugging and profiling

Debugging

- Python has its own debugger called pdb. It is a part of the Python distribution.
- pudb is a console-based Python debugger which can easily be installed using pip.
- If you are looking for IDEs with debugging capabilities, see the Editors and IDEs section.
- If you are using Windows, Python Tools for Visual Studio adds Python support for Visual Studio.
- If you would like to integrate pdb with `vim`, you can use Pyclewn.
- List of other available software can be found on the Python wiki page on debugging tools.
- If you are looking for some tutorials to get started:
  - https://pymotw.com/2/pdb
  - https://github.com/spiside/pdb-tutorial
  - https://www.jetbrains.com/help/pycharm/2016.3/debugging.html
  - https://waterprogramming.wordpress.com/2015/09/10/debugging-in-python-using-pycharm/
  - http://www.pydev.org/manual_101_run.html

Profiling

There are a number of available profiling tools that are suitable for different situations.

- cProfile measures number of function calls and how much CPU time they take. The output can be further analyzed using the `pstats` module.
- For more fine-grained, line-by-line CPU time profiling, two modules can be used:
  - line_profiler provides a function decorator that measures the time spent on each line inside the function.
  - pprofile is less intrusive; it simply times entire Python scripts line-by-line. It can give output in callgrind format, which allows you to study the statistics and call tree in `kcachegrind` (often used for analyzing c(++) profiles from `valgrind`).

More realistic profiling information can usually be obtained by using statistical or sampling profilers. The profilers listed below all create nice flame graphs.

- vprof
- Pyflame
- nylas-perftools

Logging

- logging module is the most commonly used tool to track events in Python code.
- Tutorials:
  - Official Python Logging Tutorial
  - http://docs.python-guide.org/en/latest/writing/logging
  - Python logging best practices

Writing Documentation

Python uses **Docstrings** for function level documentation. You can read a detailed description of docstring usage in PEP 257. The default location to put HTML documentation is Read the Docs. You can connect your account at Read the Docs to your GitHub account and let the HTML be generated automatically using Sphinx.

Autogenerating the documentation

There are several tools that automatically generate documentation from docstrings. At the eScience Center, we mostly use Sphinx, which uses reStructuredText as its markup language, but can be extended to use Markdown as well.

- Sphinx quickstart
- reStructuredText Primer
- Instead of using reST, Sphinx can also generate documentation from the more readable NumPy style or Google style docstrings. The Napoleon extension needs to be enabled.

We recommend using the Google documentation style. Use `sphinx-build` to build your documentation.

You can also integrate entire Jupyter notebooks into your HTML Sphinx output with nbsphinx. This way, your demo notebooks, for instance, can double as documentation. Of course, the notebooks will not be interactive in the compiled HTMl, but they will include all code and output cells.

Recommended additional packages and libraries

General scientific

- NumPy
- SciPy
- Pandas data analysis toolkit
- scikit-learn: machine learning in Python
- Cython speed up Python code by using C types and calling C functions
- dask larger than memory arrays and parallel execution

IPython and Jupyter notebooks (aka IPython notebooks)

IPython is an interactive Python interpreter -- very much the same as the standard Python interactive interpreter, but with some extra features (tab completion, shell commands, in-line help, etc).

Jupyter notebooks (formerly know as IPython notebooks) are browser based interactive Python enviroments. It incorporates the same features as the IPython console, plus some extras like in-line plotting. Look at some examples to find out more. Within a notebook you can alternate code with Markdown comments (and even LaTeX), which is great for reproducible research. Notebook extensions adds extra functionalities to notebooks. JupyterLab is a web-based environment with a lot of improvements and integrated tools.

Jupyter notebooks contain data that makes it hard to nicely keep track of code changes using version control. If you are using git, you can add filters that automatically remove output cells and unneeded metadata from your notebooks. If you do choose to keep output cells in the notebooks (which can be useful to showcase your code's capabilities statically from GitHub) use ReviewNB to automatically create nice visual diffs in your GitHub pull request threads. It is good practice to restart the kernel and run the notebook from start to finish in one go before saving and committing, so you are sure that everything works as expected.

Visualization

- Matplotlib has been the standard in scientific visualization. It supports quick-and-dirty plotting through the `pyplot` submodule. Its object oriented interface can be somewhat arcane, but is highly customizable and runs natively on many platforms, making it compatible with all major OSes and environments. It supports most sources of data, including native Python objects, Numpy and Pandas.
    - Seaborn is a Python visualisation library based on Matplotlib and aimed towards statistical analysis. It supports numpy, pandas, scipy and statmodels.
- Web-based:

  - Bokeh is Interactive Web Plotting for Python.
  - Plotly is another platform for interactive plotting through a web browser, including in Jupyter notebooks.
  - altair is a *grammar of graphics* style declarative statistical visualization library. It does not render visualizations itself, but rather outputs Vega-Lite JSON data. This can lead to a simplified workflow.
  - ggplot is a plotting library imported from R.

Parallelisation

CPython (the official and mainstream Python implementation) is not built for parallel processing due to the global interpreter lock. Note that the GIL only applies to actual Python code, so compiled modules like e.g. `numpy` do not suffer from it.

Having said that, there are many ways to run Python code in parallel:

- The multiprocessing module is the standard way to do parallel executions in one or multiple machines, it circumvents the GIL by creating multiple Python processess.
- A much simpler alternative in Python 3 is the `concurrent.futures` module.
- IPython / Jupyter notebooks have built-in parallel and distributed computing capabilities
- Many modules have parallel capabilities or can be compiled to have them.
- At the eScience Center, we have developed the Noodles package for creating computational workflows and automatically parallelizing it by dispatching independent subtasks to parallel and/or distributed systems.

Web Frameworks

There are convenient Python web frameworks available:

- flask
- CherryPy
- Django
- bottle (similar to flask, but a bit more light-weight for a JSON-REST service)

We recommend `flask`.

NLP/text mining

- nltk Natural Language Toolkit
- Pattern: web/text mining module
- gensim: Topic modeling

Creating programs with command line arguments

- For run-time configuration via command-line options, the built-in `argparse` module usually suffices.
- A more complete solution is `ConfigArgParse`. This (almost) drop-in replacement for `argparse` allows you to not only specify configuration options via command-line options, but also via (ini or yaml) configuration files and via environment variables.
- Other popular libraries are `click` and `fire`.

What is R?

Page maintainer: Malte Lüken @maltelueken

R is a functional programming language and software environment for statistical computing and graphics: https://www.r-project.org/.

Philosophy and typical use cases

R is particularly popular in the social, health, and biological sciences where it is used for statistical modeling. R can also be used for signal processing (e.g. FFT), machine learning, image analyses, and natural language processing. The R syntax is similar in compactness and readability as python and matlab by which it serves as a good prototyping environment in science.

One of the strengths of R is the large number of available open source statistical packages, often developed by domain experts. For example, R-package Seewave is specialised in sound analyses. Packages are typically released on CRAN The Comprehensive R Archive Network.

A few remarks for readers familiar with Python:

- Compared with Python, R does not need a notebook to program interactively. In RStudio, an IDE that is installed separately, the user can run sections of the code by selecting them and pressing Ctrl+Enter. Consequently the user can quickly transition from working with scripts to working interactively using the Ctrl+Enter.
- Numbering in R starts with 1 and not with 0.

Recommended sources of information

Some R packages have their own google.group. All R functions come with documentation in a standardized format. To learn R see the following resources:

- R for Data Science by Hadley Wickham,
- Advanced R by Hadley Wickham,
- Writing better R code by Laurent Gatto.

Further, stackoverflow and standard search engines can lead you to answers to issues.

Getting started

Setting up R

To install R check detailed description at CRAN website.

IDE

R programs can be written in any text editor. R code can be run from the command line or interactively within R environment, that can be started with `R` command in the shell. To quit R environment type `q()` .

RStudio is a free powerful integrated development environment (IDE) for R. It features editor with code completion, command line environment, file manager, package manager and history lookup among others. You will have to install RStudio in addition to installing R. Please note that updating RStudio does not automatically update R and the other way around.

Within RStudio you can work on ad-hoc code or create a project. Compared with Python an R project is a bit like a virtual environment as it preserves the workspace and installed packages for that project. Creating a project is needed to build an R package. A project is created via the menu at the top of the screen.

Installing compilers and runtimes

Not needed as most functions in R are already compiled in C, nevertheless R has compiling functionality as described in the R manual. See overview by Hadley Wickham.

Coding style conventions

It is good to follow the R style conventions as posted by Hadley Wickham, which is seems compatible with the R style convention as posted by Google.

One point in both style conventions that has resulted in some discussion is the '<-' syntax for variable assignment. In the majority of R tutorials and books you will see that authors use this syntax, e.g. 'a <- 3' to assign value 3 to object 'a'. Please note that R syntax 'a = 3' will preform exactly the same operation in 99.9% of situations. The = syntax has less keystrokes and could therefore be considered more efficient and readable. Further, the = syntax avoids the risk for typos like a < -1, which will produce a boolean if 'a' exists, and a <- 1 which will produce an object 'a' with a numeric value. Further, the = syntax may be more natural for those who already use it in other computing languages.

The difference between '<-' and '=' is mainly related to scoping. See the official R definition for more information. The example below demonstrates the difference in behaviour:

Define a simple function named addone to add 1 to the function input:

- addone = function(x) return(x + 1)
- addone(3)
  - will produce 4
- addone(b=3)
  - will throw an error message because the function does not know argument b
- addone(b<-3)
  - will produce 4 as it will first assign 3 to b and then uses b as value for the first argument in addone, which happens to be x
- addone(x=3)
  - will produce 4 as it will assign 3 to known function argument x

The <- supporters will argue that this example demonstrates that = should be avoided. However, it also demonstrates that = syntax can work in the context of function input if = is only used for assigning values to input arguments that are expected by the function (x in the example above) and to never introduce new R objects as part of a function call (b in the example above).

From a computer science perspective it is probably best to adhere to the <- convention. From a domain science perspective it is understandable to use =. The code performs exactly the same and guarantees that new objects created as part of a function call result in an error. Please note that it is also possible to develop code with = syntax and to transfer it to <- syntax once the code is finished, the formatR package offers tools for doing this. The CRAN repository for R packages accepts both forms of syntax.

Recommended additional packages and libraries

Plotting with basic functions and ggplot2 and ggvis

For a generic impression of what R can do see: https://www.r-graph-gallery.com/all-graphs

The basic R installation comes with a wide range of functions to plot data to a window on your screen or to a file. If you need to quickly inspect your data or create a custom-made static plot then the basic functions offer the building blocks to do the job. There is a Statmethods.net tutorial with some examples of plotting options in R.

However, externally contributed plotting packages may offer easier syntax or convenient templates for creating plots. The most popular and powerful contributed graphics package is ggplot2. Interactive plots can be made with ggvis package and embeded in web application, and this tutorial.

In summary, it is good to familiarize yourself with both the basic plotting functions as well as the contributed graphics packages. In theory, the basic plot functions can do everything that ggplot2 can do, it is mostly a matter of how much you like either syntax and how much freedom you need to tailor the visualisation to your use case.

Building interactive web applications with shiny

Thanks to shiny.app it is possible to make interactive web application in R without the need to write javascript or html.

Building reports with knitr

knitr is an R package designed to build dynamic reports in R. It's possible to generate on the fly new pdf or html documents with results of computations embedded inside.

Preparing data for analysis

There are packages that ease tidying up messy data, e.g. tidyr and reshape2. The idea of tidy and messy data is explained in a tidy data paper by Hadley Wickham. There is also the google group manipulatr to discuss topics related to data manipulation in R.

Speeding up code

As in many computing languages loops should be avoided in R. Here is a list of tricks to speed up your code:

- read.table() is sometimes faster than read.csv()
- ifelse()
- lapply()
- sapply()
- mapply()
- grep()
- %in% for testing whether and where values in one object occur in another object
- aggregate()
- which() for identifying which object indices match a certain condition
- table() for getting a frequency table of categorical data
- grep()
- gsub()
- dplyr package, see also

Use ?functionname to access fucntion documentation.

Package development

Building R packages

There is a great tutorial written by Hadley Wickam describing all the nitty gritty of building your own package in R. It's called R packages.

Package documentation

Read Documentation chapter of Hadleys R packages book for details about documenting R code.

Customary R uses `.Rd` files in `/man` directory for documentation. These files and folders are automatically created by RStudio when you create a new project from your existing R-function files.

If you use 'roxygen' function level comments starting with `#'` are recognised by `roxygen` and are used to automatically generate .Rd files. Read more about `roxygen` syntax on it's github page. `roxygen` will also populate `NAMESPACE` file which is necessary to manage package level imports.

R function documentation offers plenty of space to document the functionality, including code examples, literature references, and links to related functions. Nevertheless, it can sometimes be helpful for the user to also have a more generic description of the package with for example use-cases. You can do this with a `vignette`. Read more about vignettes in Package documentation chapter of Hadleys R packages book.

Available templates

- https://rapporter.github.io/rapport/
- https://shiny.posit.co/r/articles/build/templates/
- https://bookdown.org/yihui/rmarkdown/document-templates.html

Testing, Checking, Debugging and Profiling

Testing and checking

Testthat is a testing package by Hadley Wickham. Testing chapter of a book R packages describes in detail testing process in R with use of `testthat`. Further, testthat: Get Started with Testing by Whickham may also provide a good starting point.

See also checking and testing R packages. note that within RStudio R package check and R package test can be done via simple toolbar clicks.

Continuous integration

Continuous integration should be done with an online service.

Debugging and Profiling

Debugging is possible in RStudio, see link. For profiling tips see link

Not in this tutorial yet:

- Logging

C and C++

Page maintainer: Johan Hidding @jhidding

C++ is one of the hardest languages to learn. Entering a project where C++ coding is needed should not be taken lightly. This guide focusses on tools and documentation for use of C++ in an open-source environment.

Standards

The latest ratified standard of C++ is C++17. The first standardised version of C++ is from 1998. The next version of C++ is scheduled for 2020. With these updates (especially the 2011 one) the preferred style of C++ changed drastically. As a result, a program written in 1998 looks very different from one from 2018, but it still compiles. There are many videos on Youtube describing some of these changes and how they can be used to make your code look better (i.e. more maintainable). This goes with a warning: Don't try to be too smart; other people still have to understand your code.

Practical use

Compilers

There are two main-stream open-source C++ compilers.

- GCC
- LLVM - CLANG

Overall, these compilers are more or less similar in terms of features, language support, compile times and (perhaps most importantly) performance of the generated binaries. The generated binary performance does differ for specific algorithms. See for instance this Phoronix benchmark for a comparison of GCC 9 and Clang 7/8.

MacOS (XCode) has a custom branch of `clang`, which misses some features like OpenMP support, and its own libcxx, which misses some standard library things like the very useful `std::filesystem` module. It is nevertheless recommended to use it as much as possible to maintain binary compatibility with the rest of macOS.

If you need every last erg of performance, some cluster environments have the Intel compiler installed.

These compilers come with a lot of options. Some basic literacy in GCC and CLANG:

- `-O` changes optimisation levels

- `-std=c++xx` sets the C++ standard used
- `-I*path*` add path to search for include files
- `-o*file*` output file
- `-c` only compile, do not link
- `-Wall` be more verbose with warnings

And linker flags:

- `-l*library*` links to a library
- `-L*path*` add path to search for libraries
- `-shared` make a shared library
- `-Wl,-z,defs` ensures all symbols are accounted for when linking to a shared object

Interpreter

There **is** a C++ interpreter called Cling. This also comes with a Jupyter notebook kernel.

Build systems

There are several build systems that handle C/C++. Currently, the CMake system is most popular. It is not actually a build system itself; it generates build files based on (in theory) platform-independent and compiler-independent configuration files. It can generate Makefiles, but also Ninja files, which gives much faster build times, NMake files for Windows and more. Some popular IDEs keep automatic count for CMake, or are even completely built around it (CLion). The major drawback of CMake is the confusing documentation, but this is generally made up for in terms of community support. When Googling for ways to write your CMake files, make sure you look for "modern CMake", which is a style that has been gaining traction in the last few years and makes everything better (e.g. dependency management, but also just the CMake files themselves).

Traditionally, the auto-tools suite (AutoConf and AutoMake) was *the* way to build things on Unix; you'll probably know the three command salute:

```markup
> ./configure --prefix=~/.local
    ...
> make -j4
    ...
> make install
```

With either one of these two (CMake or Autotools), any moderately experienced user should be able to compile your code (if it compiles).

There are many other systems. Microsoft Visual Studio has its own project model / build system and a library like Qt also forces its own build system on you. We do not recommend these if you don't also supply an option for building with CMake or Autotools. Another modern alternative that has been gaining attention mainly in the GNU/Gnome/Linux world is Meson, which is also based on Ninja.

Package management

There is no standard package manager like `pip` , `npm` or `gem` for C++. This means that you will have to choose depending on your particular circumstances what tool to use for installing libraries and, possibly, packaging the tools you yourself built. Some important factors include:

- Whether or not you have root/admin access to your system
- What kind of environment/ecosystem you are working in. For instance:
  - There are many tools targeted specifically at HPC/cluster environments.
  - Specific communities (e.g. NLP research or bioinformatics) may have gravitated towards specific tools, so you'll probably want to use those for maximum impact.
- Whether software is packaged at all; many C/C++ tools only come in source form, hopefully with build setup configuration.

Yes root access

If you have root/admin access to your system, the first go-to for libraries may be your OS package manager. If the target package is not in there, try to see if there is an equivalent library that is, and see what kind of software uses it.

No root access

A good, cross-platform option nowadays is to use `miniconda` , which works on Linux, macOS and Windows. The `conda-forge` channel especially has a lot of C++ libraries. Specify that you want to use this channel with command line option `-c conda-forge` . The `bioconda` channel in turn builds upon the `conda-forge` libraries, hosting a lot of bioinformatics tools.

Managing non-packaged software

If you do have to install a programm, which depends on a specific version of a library which depends on a specific version of another library, you enter what is called *dependency hell*. Some agility in compiling and installing libraries is essential.

You can install libraries in `/usr/local` or in `${HOME}/.local` if you aren't root, but there you have no package management.

Many HPC administrations provide environment modules (`module avail`), which allow you to easily populate your `$PATH` and other environment variables to find the respective package. You can also write your own module files to solve your *dependency hell*.

A lot of libraries come with a package description for `pkg-config` . These descriptions are installed in `/usr/lib/pkgconfig` . You can point `pkg-config` to your additional libraries by setting the `PKG_CONFIG_PATH` environment variable. This also helps for instance when trying to automatically locate dependencies from CMake, which has `pkg-config` support as a fallback for when libraries don't support CMake's `find_package` .

If you want to keep things organized on systems where you use multiple versions of the same software for different projects, a simple solution is to use something like `xstow` . XStow is a poor-mans package manager. You install each library in its own directory (`~/.local/pkg/<package>` for instance), then running `xstow` will create symlinks to the files in the `~/.local` directory (one above the XStow package directory). Using XStow in this way alows you to keep a single additional search path when compiling your next library.

Packaging software

In case you find the manual compilation too cumbersome, or want to conveniently distribute software (your own or perhaps one of your project's dependencies that the author did not package themselves), you'll have to build your own package. The above solutions are good defaults for this, but there are some additional options that are widely used.

- For distribution to root/admin users: system package managers (Linux: `apt` , `yum` , `pacman` , macOS: Homebrew, Macports)
- For distribution to any users: Conda and Conan are cross-platform (Linux, macOS, Windows)
- For distribution to HPC/cluster users: see options below

When choosing which system to build your package for, it is imporant to consider your target audience. If any of these tools are already widely used in your audience, pick that one. If not, it is really up to your personal preferences, as all tools have their pros and cons. Some general guidelines could be:

- prefer multi-platform over single platform
- prefer widely used over obscure (even if it's technically magnificent, if nobody uses it, it's useless for distributing your software)

- prefer multi-language over single language (especially for C++, because it is so often used to build libraries that power higher level languages)

But, as the state of the package management ecosystem shows, in practice, there will be many exceptions to these guidelines.

HPC/cluster environments

One way around this if the system does use `module` is to use Easybuild, which makes installing modules in your home directory quite easy. Many recipes (called Easyblocks) for building packages or whole toolchains are available online. These are written in Python.

A similar package that is used a lot in the bioinformatics community is guix. With guix, you can create virtual environments, much like those in Python `virtualenv` or Conda. You can also create relocatable binaries to use your binaries on systems that do not have guix installed. This makes it easy to test your packages on your laptop before deploying to a cluster system.

A package that gains more traction at the moment for HPC environments is spack. Spack allows you to pick from many compilers. When installing packages, it compiles every package from scratch. This allows you to be tailor compilation flags and such to take fullest advantage of your cluster's hardware, which can be essential in HPC situations

Near future: Modules

Note that C++20 will bring Modules, which can be used as an alternative to including (precompiled) header files. This will allow for easier packaging and will probably cause the package management landscape to change considerably. For this reason, it may be wise at this time to keep your options open and keep an eye on developments within the different package management solutions.

Editors

This is largely a matter of taste, but not always.

In theory, given that there are many good command line tools available for working with C(++) code, any code editor will do to write C(++). Some people also prefer to avoid relying on IDEs too much; by helping your memory they can also help you to write less maintainable code. People of this persuasion would usually recommend any of the following editors:

- Vim, recommended plugins:
    - NERDTree file explorer.

    - editorconfig
    - stl.vim adds STL to syntax highlighting
    - Syntastic
    - Integrated debugging using Clewn
- Emacs:
    - Has GDB mode for debugging.
- More modern editors: Atom / Sublime Text / VS Code
    - Rich plugin ecosystem
    - Easier on the eyes… I mean modern OS/GUI integration

In practice, sometimes you run into large/complex existing projects and navigating these can be really hard, especially when you just start working on the project. In these cases, an IDE can really help. Intelligent code suggestions, easy jumping between code segments in different files, integrated debugging, testing, VCS, etc. can make the learning curve a lot less steep. Good/popular IDEs are

- CLion
- Visual Studio (Windows only, but many people swear by it)
- Eclipse

Code and program quality analysis

C++ (and C) compilers come with built in linters and tools to check that your program runs correctly, make sure you use those. In order to find issues, it is probably a good idea to use both compilers (and maybe the valgrind memcheck tool too), because they tend to detect different problems.

Automatic Formatting with clang-format

While most IDEs and some editors offer automatic formatting of files, clang-format is a standalone tool, which offers sensible defaults and a huge range of customisation options. Integrating it into the CI workflow guarantees that checked in code adheres to formatting guidelines.

Static code analysis with GCC

To use the GCC linter, use the following set of compiler flags when compiling C++ code:

```
-O2 -Wall -Wextra -Wcast-align -Wcast-qual -Wctor-dtor-privacy -Wdisabled-optimization
-Wformat=2
-Winit-self -Wlogical-op -Wmissing-declarations -Wmissing-include-dirs -Wnoexcept -
Wold-style-cast
```

```
-Woverloaded-virtual -Wredundant-decls -Wshadow -Wsign-conversion -Wsign-promo -
Wstrict-null-sentinel
-Wstrict-overflow=5 -Wswitch-default -Wundef -Wno-unused
```

and these flags when compiling C code:

```
-O2 -Wall -Wextra -Wformat-nonliteral -Wcast-align -Wpointer-arith -Wbad-function-cast
-Wmissing-prototypes -Wstrict-prototypes -Wmissing-declarations -Winline -Wundef
-Wnested-externs -Wcast-qual -Wshadow -Wwrite-strings -Wno-unused-parameter
-Wfloat-equal
```

Use at least optimization level 2 (`-O2`) to have GCC perform code analysis up to a level where you get all warnings. Use the `-Werror` flag to turn warnings into errors, i.e. your code won't compile if you have warnings. See this post for an explanation of why this is a reasonable selection of warning flags.

Static code analysis with Clang (LLVM)

Clang has the very convenient flag

```
-Weverything
```

A good strategy is probably to start out using this flag and then disable any warnings that you do not find useful.

Static code analysis with cppcheck

An additional good tool that detects many issues is cppcheck. Most editors/IDEs have plugins to use it automatically.

Dynamic program analysis using `-fsanitize`

Both GCC and Clang allow you to compile your code with the `-fsanitize=` flag, which will instrument your program to detect various errors quickly. The most useful option is probably

```
-fsanitize=address -O2 -fno-omit-frame-pointer -g
```

which is a fast memory error detector. There are also other options available like `-fsanitize=thread` and `-fsanitize=undefined`. See the GCC man page or the Clang online manual for more information.

Dynamic program analysis using the valgrind suite of tools

The valgrind suite of tools has tools similar to what is provided by the `-fsanitize` compiler flag as well as various profiling tools. Using the valgrind tool memcheck to detect memory errors is typically slower than using compiler provided option, so this might be something you will want to do less often. You will probably want to compile your code with debug symbols enabled (`-g`) in order to get useful output with memcheck. When using the profilers, keep in mind that a statistical profiler may give you more realistic results.

Automated code refactoring

Sometimes you have to update large parts of your code base a little bit, like when you move from one standard to another or you changed a function definition. Although this can be accomplished with a `sed` command using regular expressions, this approach is dangerous, if you use macros, your code is not formatted properly etc.... Clang-tidy can do these things and many more by using the abstract syntax tree of the compiler instead of the source code files to refactor your code and thus is much more robust but also powerful.

Debugging

Most of your time programming C(++) will probably be spent on debugging. At some point, surrounding every line of your code with `printf("here %d", i++);` will no longer avail you and you will need a more powerful tool. With a debugger, you can inspect the program while it is running. You can pause it, either at random points when you feel like it or, more usually, at so-called breakpoints that you specified in advance, for instance at a certain line in your code, or when a certain function is called. When paused, you can inspect the current values of variables, manually step forward in the code line by line (or by function, or to the next breakpoint) and even change values and continue running. Learning to use these powerful tools is a very good time investment. There are some really good CppCon videos about debugging on YouTube.

- GDB - the GNU Debugger, many graphical front-ends are based on GDB.
- LLDB - the LLVM debugger. This is the go-to GDB alternative for the LLVM toolchain, especially on macOS where GDB is hard to setup.
- DDD - primitive GUI frontend for GDB.
- The IDEs mentioned above either have custom built-in debuggers or provide an interface to GDB or LLDB.

Libraries

Historically, many C and C++ projects have seemed rather hestitant about using external dependencies (perhaps due to the poor dependency management situation mentioned above). However, many good (scientific) computing libraries are available today that you should consider using if applicable. Here follows a list of libraries that we recommend and/or have experience with. These can typically be installed from a wide range of package managers.

Usual suspects

These scientific libraries are well known, widely used and have a lot of good online documentation.

- GNU Scientific library (GSL)
- FFTW: Fastest Fourier Transform in the West
- OpenMPI. Use with caution, since it will strongly define the structure of your code, which may or may not be desirable.

Boost

This is what the Google style guide has to say about Boost:

> - **Definition:** The Boost library collection is a popular collection of peer-reviewed, free, open-source C++ libraries.
> - **Pros:** Boost code is generally very high-quality, is widely portable, and fills many important gaps in the C++ standard library, such as type traits and better binders.
> - **Cons:** Some Boost libraries encourage coding practices which can hamper readability, such as metaprogramming and other advanced template techniques, and an excessively "functional" style of programming.

As a general rule, don't use Boost when there is equivalent STL functionality.

xtensor

xtensor is a modern (C++14) N-dimensional tensor (array, matrix, etc) library for numerical work in the style of Python's NumPy. It aims for maximum performance (and in most cases it succeeds) and has an active development community. This library features, among other things:

- Lazy-evaluation: only calculate when necessary.
- Extensible template expressions: automatically optimize many subsequent operations into one "kernel".
- NumPy style syntax, including broadcasting.

- C++ STL style interfaces for easy integration with STL functionality.
- Very low-effort integration with today's main data science languages Python, R and Julia. This all makes xtensor a very interesting choice compared to similar older libraries like Eigen and Armadillo.

General purpose, I/O

- Configuration file reading and writing:
    - yaml-cpp: A YAML parser and emitter in C++
    - JSON for Modern C++
- Command line argument parsing:
    - argagg
    - Clara
- fmt: pythonic string formatting
- hdf5-cpp: The popular HDF5 binary format C++ interface.

Parallel processing

- oneAPI Threading Building Blocks (oneTBB): template library for task parallelism
- ZeroMQ: lower level flexible communication library with a unified interface for message passing between threads and processes, but also between separate machines via TCP.

Style

Style guides

Good style is not just about layout and linting on trailing whitespace. It will mean the difference between a blazing fast code and a broken one.

- C++ Core Guidelines
- Guidelines Support Library
- Google Style Guide
- Google Style Guide - github Contains the CppLint linter.

Project layout

A C++ project will usually have directories `/src` for source codes, `/doc` for Doxygen output, `/test` for testing code. Some people like to put header files in `/include`. In C++ though, many header files will contain functioning code (templates and inline functions). This makes the separation between code and interface a bit murky. In this case, it can make more sense to put headers and implementation in the same tree, but different communities will have different opinions on this. A third option that is sometimes used is to make separate "template implementation" header files.

Sustainability

Testing

Use Google Test. It is light-weight, good and is used a lot. Catch2 is also pretty good, well maintained and has native support in the CLion IDE.

Documentation

Use Doxygen. It is the de-facto standard way of inlining documentation into comment sections of your code. The output is very ugly. Mini-tutorial: run `doxygen -g` (preferably inside a `doc` folder) in a new project to set things up, from then on, run `doxygen` to (re-)generate the documentation.

A newer but less mature option is cldoc.

Resources

Online

- CppCon videos: Many really good talks recorded at the various CppCon meetings.
- CppReference.com
- C++ Annotations
- CPlusPlus.com
- Modern C++, according to Microsoft

Books

- Bjarne Soustrup - The C++ Language
- Scott Meyers - Effective Modern C++

Fortran

Page maintainer: Gijs van den Oord @goord

Disclaimer: In general the Netherlands eScience Center does not recommend using Fortran. However, in some cases it is the only viable option, for instance if a project builds upon existing code written in this language. This section will be restricted to Fortran90, which captures majority of Fortran source code.

The second use case may be extremely performance-critical dense numerical compute workloads, with no existing alternative. In this case it is recommended to keep the Fortran part of the application minimal, using a high-level language like Python for program control flow, IO, and user interface.

Recommended sources of information

- Fortran90 official documentation
- Fortran wiki
- Fortran90 handbook

Compilers

- **gfortran**: the official GNU Fortran compiler and part of the gcc compiler suite.
- **ifort**: the Intel Fortran compiler, widely used in academia and industry because of its superior performance, but unfortunately this is commercial software so not recommended. The same holds for the Portland compiler **pgfortran**

Debuggers and diagnostic tools

There exist many commercial performance profiling tools by Intel and the Portland Group which we shall not discuss here. Most important freely available alternatives are

- **gdb**: the GNU debugger, part of the gcc compiler suite. Use the **-g** option to compile with debugging symbols.
- **gprof**: the GNU profiler, part of gcc too. Use the **-p** option to compile with profiling enabled.
- **valgrind**: to detect memory leaks.

Editors and IDEs

Most lightweight editors provide Fortran syntax highlighting. Vim and emacs are most widely used, but for code completion and refactoring tools one might consider the CBFortran distribution of Code::Blocks.

Coding style conventions

If working on an existing code base, adopt the existing conventions. Otherwise we recommend the standard conventions, described in the official documentation and the Fortran company style guide. We would like to add the following advice:

- Use free-form text input style (the default), with a maximal line width well below the 132 characters imposed by the Fortran90 standard.
- When a method does not need to alter any data in any module and returns a single value, use a function for it, otherwise use a subroutine. Minimize the latter to reasonable extent.
- Use the intent attributes in subroutine variable declarations as it makes the code much easier to understand.
- Use a performance-driven approach to the architecture, do not use the object-oriented features of Fortran90 if they slow down execution. Encapsulation by modules is perfectly acceptable.
- Add concise comments to modules and routines, and add comments to less obvious lines of code.
- Provide a test suite with your code, containing both unit and integration tests. Both automake and cmake provide test suite functionality; if you create your makefile yourself, add a separate testing target.

Technology Overview

Page maintainer: Patrick Bos [@egpbos]

These chapters are based on our experiences with using specific software technologies.

The main audience is RSEs familiar with basic computing and programming concepts.

The purpose of these chapters is for someone unfamiliar with the specific technology to get a quick overview of the most important concepts, practices and tools, without going into too much detail (we provide links to further reading material for more).

GPU Programming Languages

Page maintainer: Alessio Sclocco @isazi

Learning Resources

- Carpentries GPU Programming course
  - Lesson material
- Introduction to CUDA C
  - Slides
  - Video
- Introduction to OpenACC
  - Slides
- Introduction to HIP Programming
  - Video
- SYCL Introduction and Best Practices
  - Video
- CSCS GPU Programming with Julia
  - Course recordings

Documentation

- CUDA
  - C programming guide
  - Runtime API
  - Driver API
  - Fortran programming guide
- HIP

- Kernel language syntax
- Runtime API

- SYCL
  - Specification
  - Reference guide

- OpenCL
  - Guide
  - API
  - OpenCL C specification
  - Reference guide

- OpenACC
  - Programming guide
  - Reference guide

- OpenMP
  - Reference guide

Overview of Libraries

- CUDA
  - cuBLAS
  - NVBLAS
  - cuFFT
  - cuGRAPH
  - cuRAND
  - cuSPARSE
- HIP
  - hipBLAS
  - hipFFT
  - hipRAND
  - hipSPARSE
- SYCL
  - OneAPI BLAS
  - OneAPI FFT

    - OneAPI sparse
    - OneAPI random number generators
- OpenCL
    - CLBlast
    - clFFT

Source-to-source Translation

- CUDA to HIP
    - hipify
- CUDA to SYCL
    - SYCLomatic
- CUDA to OpenCL
    - cutocl

Foreign Function Interfaces

- C++
    - CUDA
        - cudawrappers
    - OpenCL
        - CLHPP
- Python
    - CUDA
        - PyCuda
        - CuPy
        - cuda-python
    - HIP
        - PyHIP
    - SYCL
        - dpctl
    - OpenCL
        - PyOpenCL
- Julia
    - CUDA
        - CUDA.jl
    - HIP
        - AMDGPU.jl

    - SYCL
        - oneAPI.jl
- Java
    - CUDA
        - JCuda
    - OpenCL
        - JOCL

High-Level Abstractions

- C++
    - Kokkos
    - Raja
- Python
    - Numba
    - pykokkos

Debugging and Profiling Tools

- CUDA
    - Nsight Systems
    - Nsight Compute
    - CUDA-GDB
    - compute-sanitizer
- HIP
    - omniperf
    - rocprof
- SYCL
    - oneprof
    - onetrace

Performance Optimization

- PRACE best practice guide on modern accelerators
- CUDA best practices
- OneAPI SYCL best practices

Auto-tuning

- Kernel Tuner
  - GitHub repository
  - Documentation
  - Tutorial

User Experience (UX)

Page maintainer: Jesus Garcia @ctwhome

User Experience Design (UX) is a broad, holistic science that combines many cognitive and brain sciences disciplines like psychology and sociology, content strategies, and arts and aesthetics by following human-center approaches.

> **Human-centred design is an approach to interactive systems development that aims to make systems usable and useful by focusing on the users, their needs and requirements, and applying human factors/ergonomics and usability knowledge and techniques. This approach enhances effectiveness and efficiency, improves human well-being, user satisfaction, accessibility, sustainability, and counteracts possible adverse effects on human health, safety, and performance. HCDSociety**

Table of content

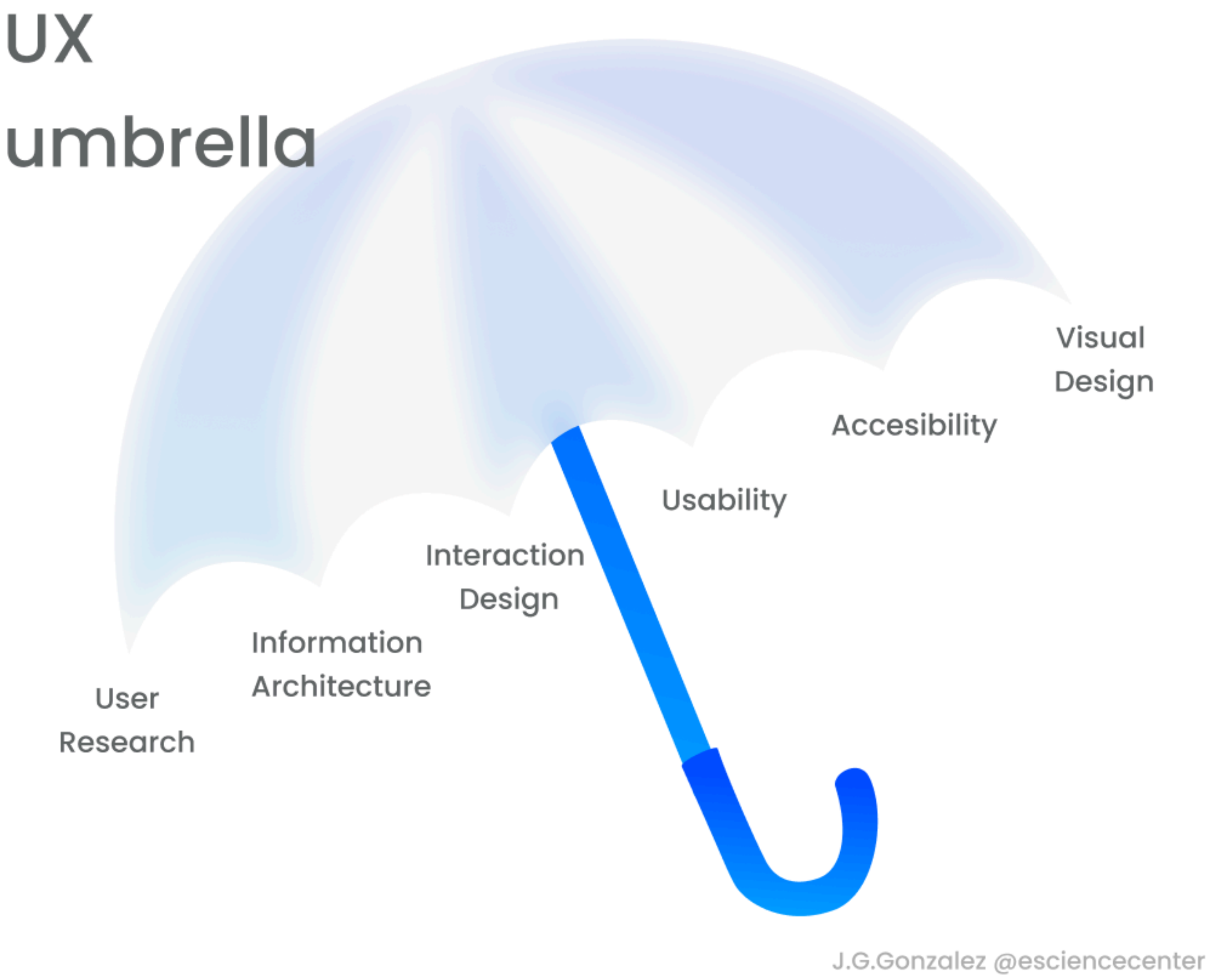
UX disciplines

The principles and indications taught by interaction-design.org can be useful in the process of creating research software.

The main UX disciplines are:

1. **User research**: understanding the people who use a product or system through observations.
2. **Information architecture**: identifying and organizing information within a system in a purposeful and meaningful way.
3. **Interaction design**: designing a product or system's interactive behaviors with a specific focus on their use.
4. **Usability evaluation**: measuring the quality of a user's experience when interacting with a product or system.
5. **Accessibility evaluation:** measuring the quality of a product or system to be accessed irrespective of personal abilities and device properties.
6. **Visual design**: designing the visual attributes of a product or system in an aesthetically pleasing way.

The known UX umbrella diagram represents the different disciplines of UX:

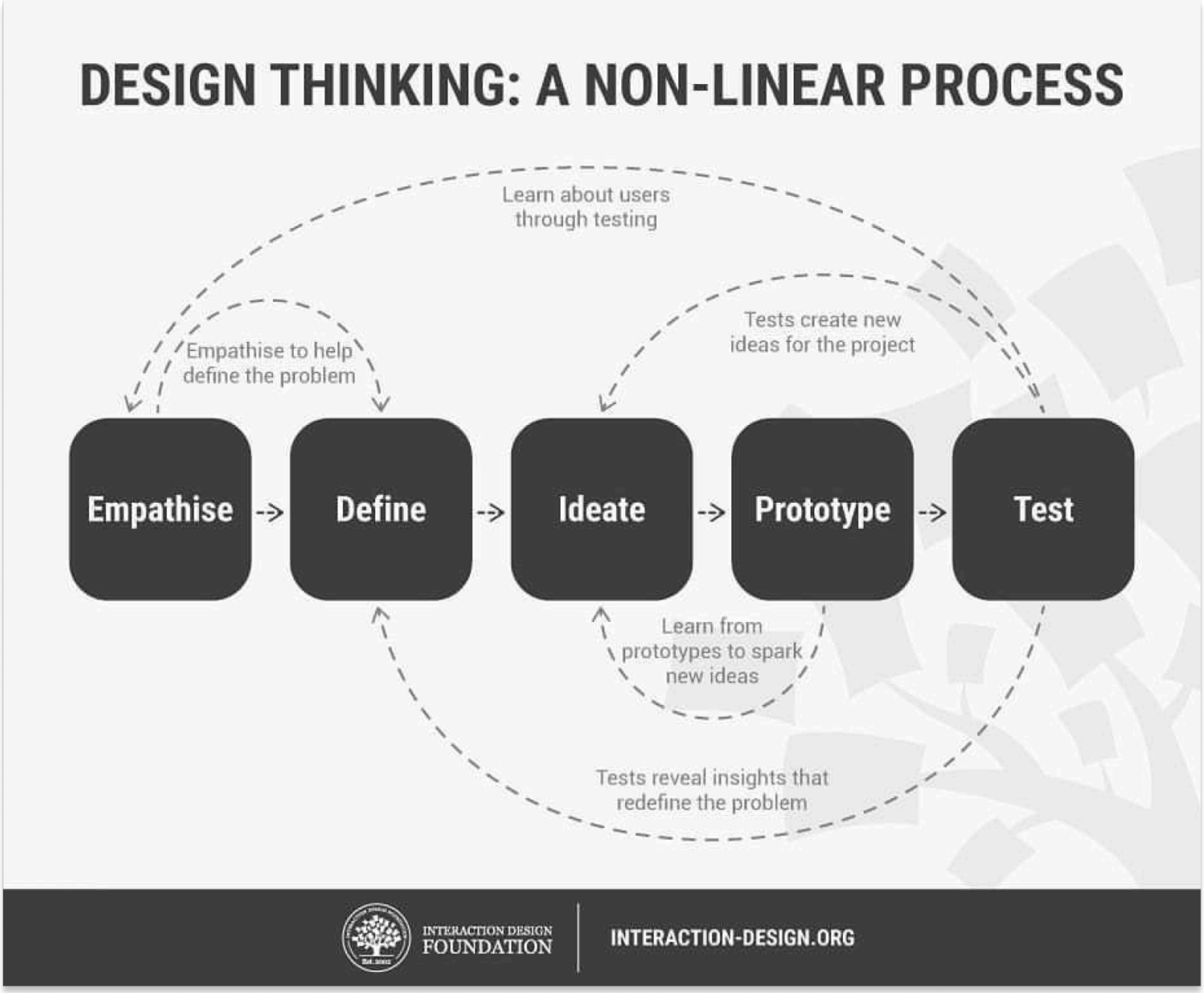


Author/Copyright holder: J.G. Gonzalez and The Netherlands eScience Center. Copyright: Apache License 2.0

Design Thinking

Design thinking is an approach, mindset, or ideology for product development. According to the IxF(Interaction Design Foundation, Design thinking achieves all these advantages at the same time:

- It is a user-centered process that starts with user data, creates design artifacts that address real and not imaginary user needs, and then tests those artifacts with real users.
- It leverages the collective expertise and establishes a shared language and buy-in amongst your team.
- It encourages innovation by exploring multiple avenues for the same problem.


DESIGN THINKING: A NON-LINEAR PROCESS
Learn about users through testing
Empathise to help define the problem
Tests create new ideas for the project
Empathise
Define
Ideate
Prototype
Test
Learn from prototypes to spark new ideas
Tests reveal insights that redefine the problem
INTERACTION DESIGN FOUNDATION
INTERACTION-DESIGN.ORG

Author/Copyright holder: Teo Yu Siang and Interaction Design Foundation. Copyright licence: CC BY-NC-SA 3.0

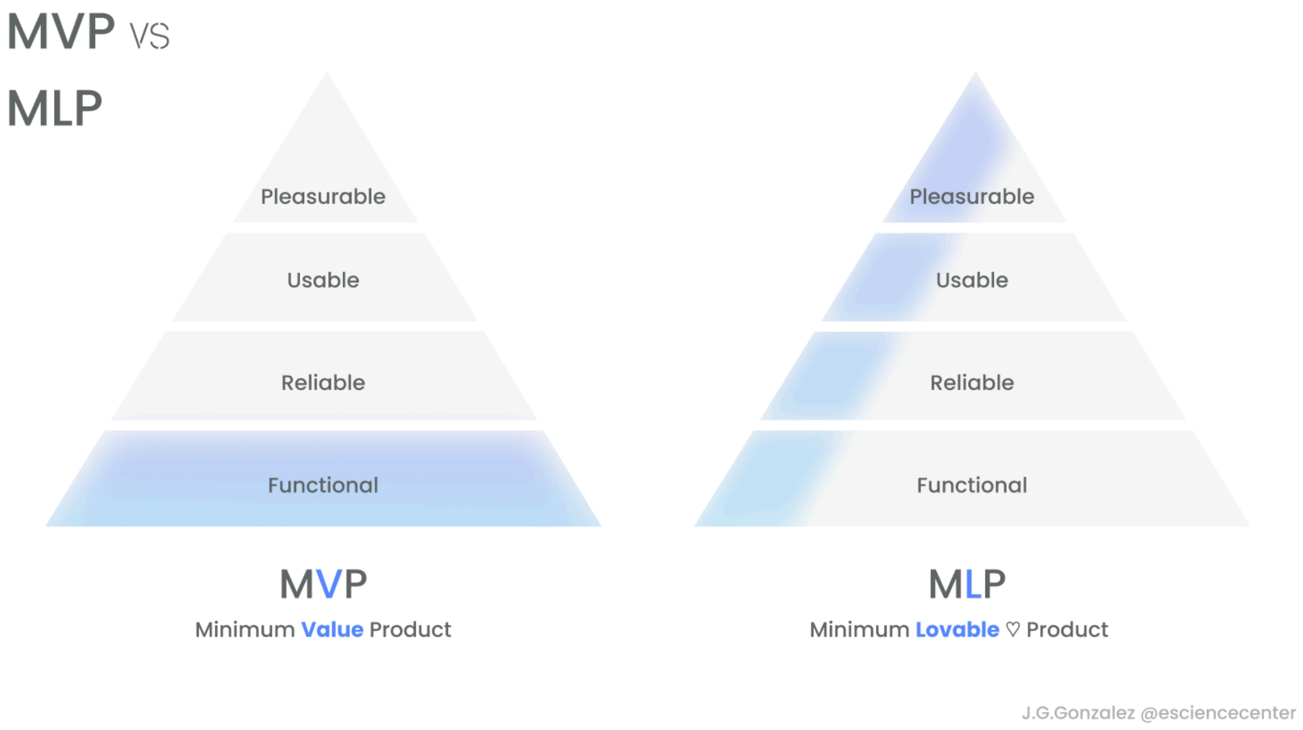
You can find more information about Design Thinking on the IxF page.

Designing software

Heuristics, or commonly known 'as the rule of thumb,' play a significant role when users interact with software. The Nielsen/Norman group has a top 10 Usability Heuristics for User Interface Design to consider when developing software.

Designing Lovable software

When delivering software iteratively, one of the common approaches to follow is to define a Minimum Value Product that contains the minimum requirements. Often is forgotten in this approach to deliver software that attracts and engages the users. When developing research software, researchers should present the new and innovative outcomes in a way that feels comfortable and easy to use from the very beginning, eliminating any cognitive burden that the software's interaction may include.



Author/Copyright holder: J.G. Gonzalez and The Netherlands eScience Center. Copyright: Apache License 2.0

While MVP (Minumun Product Value) focuses on provide users with a way to explore the product and understand its main intent, MLP (Minimun Loveable Product) approach focuses on essential features instead of the bare minimum expected from a class software. Going beyond the bare functionality, the attention is driven towards a great user experience. The outcomes mush contains all elements in the pyramid being **functional, reliable, usable, and pleasurable.**

Tools and resources

Design tools used for Visual Design, Prototyping, and IxD testing collaborative, real-time, online, and multiplatform.

- Figma
- Miro
- Whimsical

Working with tabular data

Page maintainers: Suvayu Ali @suvayu , Flavio Hafner @f-hafner and Reggie Cushing @recap

There are several solutions available to you as an RSE, with their own pros and cons. You should evaluate which one works best for your project, and project partners, and pick one. Sometimes it might be, that you need to combine two different types of technologies. Here are some examples from our experience.

You will encounter datasets in various file formats like:

- CSV/Excel
- Parquet
- HDF5/NetCDF
- JSON/JSON-LD

Or local database files like SQLite. It is important to note, the various trade-offs between these formats. For instance, doing a random seek is difficult with a large dataset for non-binary formats like: CSV, Excel, or JSON. In such cases you should consider formats like Parquet, or HDF5/NetCDF. Non-binary files can also be imported into local databases like SQLite or DuckDB. Below we compare some options to work with datasets in these formats.

It's also good to know about Apache Arrow, which is not itself a file format, but a specification for a memory layout of (binary) data. There is an ecosystem of libraries for all major languages to handle data in this format. It is used as the back-end of many data handling projects, among which a few others mentioned in this chapter.

Local database

When you have a relational dataset, it is recommended that you use a database. Using local databases like SQLite and DuckDB can be very easy because of no setup requirements. But they come with some some limitations; for instance, multiple users cannot write to the database simultaneously.

SQLite is a transactional database, so if you have a dataset that is changing with time (e.g. you are adding new rows), it would be more appropriate. However in research often we work with static databases, and are interested mostly in analytical tasks. For such a case, DuckDB is a more appropriate alternative. Between the two,

- DuckDB can also create views (virtual tables) from other sources like files, other databases, but with SQLite you always have to import the data before running any queries.

- DuckDB is multi-threaded. This can be an advantage for large databases, where aggregation queries tend to be faster than sqlite.

    - However if you have a really large dataset, say 100Ms of rows, and want to perform a deeply nested query, it would require substantial amount of memory, making it unfeasible to run on personal laptops.
    - There are options to customize memory handling, and push what is possible on a single machine.

      You need to limit the memory usage to prevent the operatings system, or shell from preemptively killing it. You can choose a value about 50% of your system's RAM.

      ```sql
      SET memory_limit = '5GB';
      ```

      By default, DuckDB spills over to disk when memory usage grows beyond the above limit. You can verify the temporary directory by running:

      ```sql
      SELECT current_setting('temp_directory') AS temp_directory;
      ```

      Note, if your query is deeply nested, you should have sufficient disk space for DuckDB to use; e.g. for 4 nested levels of `INNER JOIN` combined with a `GROUP BY` , we observed a disk spill over of 30x the original dataset. However we found this was not always reliable.

      In this kind of borderline cases, it might be possible to address the limitation by splitting the workload into chunks, and aggregating later, or by considering one of the alternatives mentioned below.

    - You can also optimize the queries for DuckDB, but that requires a deeper dive into the documentation, and understanding how DuckDB query optimisation works.

- Both databases support setting (unique) indexes. Indexes are useful and sometimes necessary
    - For both DuckDB and SQLite, unique indexes allow to ensure data integrity
    - For SQLite, indexes are crucial to improve the performance of queries. However, having more indexes makes writing new records to the database slower. So it's again a trade-off between query and write speed.

Useful libraries

Database APIs

- SQLAlchemy

- In Python, interfacing to SQL databases like SQLite, MySQL or PostgreSQL is often done using SQLAlchemy, which is an Object Relational Mapper (ORM) that allows you to map tables to Python classes. Note that you still need to use a lot of manual SQL outside of Python to manage the database. However, SQLAlchemy allows you to use the data in a Pythonic way once you have the database layout figured out.

Data processing libraries on a single machine

- Pandas
  - The standard tool for working with dataframes, and widely used in analytics or machine learning workflows. Note however how Pandas uses memory, because certain APIs create copies, while others do not. So if you are chaining multiple operations, it is preferable to use APIs that avoid copies.
- Vaex
  - Vaex is an alternative that focuses on out-of-core processing (larger than memory), and has some lazy evaluation capabilities.
- Polars
  - An alternative to Pandas (started in 2020), which is primarily written in Rust. Compared to pandas, it is multi-threaded and does lazy evaluation with query optimisation, so much more performant. However since it is newer, documentation is not as complete. It also allows you to write your own custom extensions in Rust.
- Apache Datafusion
  - A very fast, extensible query engine for building high-quality data-centric systems in Rust, using the Apache Arrow in-memory format. DataFusion offers SQL and Dataframe APIs, excellent performance, built-in support for CSV, Parquet, JSON, and Avro, extensive customization, and a great community.

Distributed/multi-node data processing libraries

- Dask
  - `dask.dataframe` and `dask.array` provides the same API as pandas and numpy respectively, making it easy to switch.
  - When working with multiple nodes, it requires communication across nodes (which is network bound).
- Ray
- Apache Spark

Software Development

In this chapter we give an overview of the best practices for software development at the Netherlands eScience Center, including a rationale.

Know your tools

In addition to the advice on the best practices in these chapters, knowing the tools that are available for software development can really help you getting things done faster.

Learn how to use the command line efficiently

Read the chapter on using Bash.

Use an editor that helps you develop

Commonly used editors and their ecosystem of plugins can really help you write better code faster. Note that for each of the editors and environments listed below, it is important to configure them such that they support the programming languages that you are developing in.

Below is a list of editors that support many programming languages.

Integrated Development Environments (IDEs):

- Visual Studio Code - modern IDE
- Atom - modern IDE
- Eclipse - a bit older but still nice

Text editors:

- Sublime Text - modern text editor
- vim - classic text editor
- emacs - classic text editor

Code Quality

Ways to improve code quality are in the Code quality chapter on the Turing Way.

There are online software quality improvement tools see the language guides for good options per language.

Editorconfig

The eScience Center has a shared editor config file

Name spaces

If your language supports namespaces, use **nl.esciencecenter** or better a namespace based on the project.

Code reviews

See the Code Reviews section.

Code reviews

Introduction

At the eScience Center, we value software quality. Higher quality software has fewer defects, better security, and better performance, which leads to happier users who can work more effectively.

Code reviews are an effective method for improving software quality. McConnell (2004) suggests that unit testing finds approximately 25% of defects, function testing 35%, integration testing 45%, and code review 55-60%. While that means that none of these methods are good enough on their own, and that they should be combined, clearly code review is an essential tool here.

Code review also improves the development process. By reviewing new additions for quality, less technical debt is accumulated, which helps long-term maintainability of the code. Reviewing lets developers learn from each other, and spreads knowledge of the code around the team. It is also a good means of getting new developers up to speed.

The main downside of code reviews is that they take time and effort. In particular, if someone from outside the project does the reviewing, they'll have to learn the code, which is a significant investment. Once up to speed, the burden is reduced significantly however, and the returns include a much smaller amount of time spent debugging later.

Approach

It's important to distinguish between semi-formal code *reviews* and formal code *inspections*. The latter involve "up to six participants and hours of meetings paging through detailed code printouts" (SMARTBEAR 2016). As this extra formality does not seem to yield better results, we limit ourselves to light-weight, informal code reviews.

Process

We haven't yet decided on how to integrate code reviews into our working process. While that gets hashed out, here is some general advice from various sources and experience.

- Review everything, nothing is too short or simple
- Try to have something else to do, and spread the load throughout your working day. Don't review full-time.
- Don't review for more than an hour at a time, after that the success rate drops quite quickly

- Don't review more than 400 lines of code (LOC) at a time, less than 200 LOC is better

- Take the time, read carefully, don't review more than 500 LOC / hour

Prerequisites

Before handing over a change or a set of code for review, the following items should be there for the reviewer to work with:

- Documentation on what was changed and why (feature, bug, issue #, etc.)
- Comments / annotations by the author on the code itself
- Test cases

Also, before doing a code review, make sure any *tools* have run that check the code automatically, e.g. checkers for coding conventions and static analysis tools, and the test suite. Ideally, these are run as part of the continuous integration infrastructure.

Review checklist

This section provides two checklists for code reviews, one for the whole program, and one for individual files or proposed changes.

In all cases, the goal is to use your brain and your programming experience to figure out how to make the code better. The lists are intended to be a source of inspiration and a description of what should be best practices in most circumstances. Some items on this list may not apply to your project or programming language, in which case they should be disregarded.

Excluded from this checklist

The following items are part of a software quality check, but are better done by an automated tool than by a human. As such, they've been excluded from this checklist. If tools are not available, they should be checked manually.

- Coding conventions (e.g. PEP 8)
- Test coverage

Rubric for assessing code quality

All code should be level 3 or 4.

Level	1	2	3	4
names	names appear unreadable, meaningless or misleading	names accurately describe the intent of the code, but can be incomplete, lengthy, misspelled or inconsistent use of casing	names accurately describe the intent of the code, and are complete, distinctive, concise, correctly spelled and consistent use of casing	all names in the program use a consistent vocabulary
headers	headers are generally missing or descriptions are redundant or obsolete; use mixed languages or are misspelled	header comments are generally present; summarize the goal of parts of the program and how to use those; but may be somewhat inaccurate or incomplete	header comments are generally present; accurately summarize the role of parts of the program and how to use those; but may still be wordy	header comments are generally present; contain only essential explanations, information and references
comments	comments are generally missing, redundant or obsolete; use mixed languages or are misspelled	comments explain code and potential problems, but may be wordy	comments explain code and potential problems, are concise	comments are only present where strictly needed
layout	old commented out code is present or lines are generally too long to read	positioning of elements within source files is not optimized for readability	positioning of elements within source files is optimized for readability	positioning of elements is consistent between files and in line with platform conventions
formatting	formatting is missing or misleading	indentation, line breaks, spacing and brackets highlight the intended structure but erratically	indentation, line breaks, spacing and brackets consistently highlight the intended structure	formatting makes similar parts of code clearly identifiable
flow	there is deep nesting; code performs more than one task per line;	flow is complex or contains many exceptions or jumps; parts of code are duplicate	flow is simple and contains few exceptions or jumps; duplication is very limited	in the case of exceptions or jumps, the most common path through the code is clearly visible

	unreachable code is present			
idiom	control structures are customized in a misleading way	choice of control structures is inappropriate	choice of control structures is appropriate; reuse of library functionality may be limited	reuse of library functionality and generic data structures where possible
expressions	expressions are repeated or contain unnamed constants	expressions are complex or long; data types are inappropriate	expressions are simple; data types are appropriate	expressions are all essential for control flow
decomposition	most code is in one or a few big routines; variables are reused for different purposes	most routines are limited in length but mix tasks; routines share many variables instead of having parameters	routines perform a limited set of tasks divided into parts; use of shared variables is limited	routines perform a very limited set of tasks and the number of parameters and shared variables is limited
modularization	most code is in one or a few large modules; or modules are artificially separated	modules have mixed responsibilities, contain many variables or contain many routines	modules have clearly defined responsibilities, contain few variables and a somewhat limited amount of routines	modules are defined such that communication between them is limited

- no need to assess a level that is not relevant to the software
- level 2 implies that the features in level 1 are not present, level 4 implies that the features in level 3 are also present

This rubric is based on:

> Stegeman, Barendsen, & Smetsers (2016). Designing a rubric for feedback on code quality in programming courses. In proceedings of the 16th Koli Calling International Conference on Computing Education Research. ACM.

Program level checklist

Here is a list of things to consider when looking at the program as a whole, rather than when looking at an individual file or change.

Documentation

Documentation is a prerequisite for using, developing and reviewing the program. Here are some things to check for.

- Is there a description of the purpose of the program or library?
- Are detailed requirements listed?
- Are requirements ranked according to MoSCoW?
- Is the use and function of third-party libraries documented?
- Is the structure/architecture of the program documented? (see below)
- Is there an installation manual?
- Is there a user manual?
- Is there documentation on how to contribute?
    - Including how to submit changes
    - Including how to document your changes

Architecture

These items are mainly important for larger programs, but may still be good to consider for small ones as well.

- Is the program split up into clearly separated modules?
- Are these modules as small as they can be?
- Is there a clear, hierarchical or layered, dependency structure between these modules?
    - If not, functionality should be rearranged, or perhaps heavily interdependent modules should be combined
- Can the design be simplified?

Security

If you're making software that is accessible to the outside world (e.g. a web application), then security becomes important. Security issues are defects, but not all defects are security issues. A security-conscious design can help mitigate the security impact of defects.

- Which modules deal with user input?
- Which modules generate output?
- Are input and output compartmentalised?
    - If not, consider making separate modules that manage all input and output, so validation can happen in one place
- In which modules is untrusted data present?
    - The fewer the better

- Is untrusted data compartmentalised?
    - Ideally, validate in the input module and pass only validated data to other parts

Legal

"I'm an engineer, not a lawyer!" is an oft-overheard phrase, but being an engineer doesn't give you permission to ignore the legal rights of the creators of the code you're using. Here are some things to check. When in doubt, ask your licensing person for advice.

- Are the licenses of all modules/libraries that are used documented?
- Are the requirements set by those licenses fulfilled?
    - Are the licenses included where needed?
    - Are copyright statements included in the code where needed?
    - Are copyright statements included in the documentation where needed?
- Are the licenses of all the parts compatible with each other?
- Is the project license compatible with all libraries?

File/Change level checklist

When you're checking individual changes (e.g. pull requests) or files, the code itself becomes the subject of scrutiny. Depending on the language, files may contain interfaces, classes or other type definitions, and functions. All these should be checked, as well as the file overall:

- Does this file contain a logical grouping of functionality?
- How big is it? Should it be split up?
- Is it easy to understand?
- Can any of the code be replaced by library functions?

Interfaces

- Is the interface documented?
- Does the concept it models make sense?
- Can it be split up further? (Interfaces should be as small as possible)

Note that most of the following items assume an object-oriented programming style, which may not be relevant to the code you're looking at.

Classes and types

- Is the class documented?

- Does it have a single responsibility? Can it be split?
- If it's designed to be extended, can it be?
- If it's not designed to be extended, is it protected against that? (e.g. final declarations)
- If it's derived from another class, can you substitute an object of this class for one of its parent class(es)?
- Is the class testable?
  - Are the dependencies clear and explicit?
  - Does it have a small number of dependencies?
  - Does it depend on interfaces, rather than on classes?

Function/Method declarations

- Are there comments that describe the intent of the function or method?
- Are input and output documented? Including units?
- Are pre- and postconditions documented?
- Are edge cases and unusual things commented?

Function/Method definitions

- Are edge cases and unusual things commented?
- Is there incomplete code?
- Could this function be split up (is it not too long)?
- Does it work? Perform intended function, logic correct, ...
- Is it easy to understand?
- Is there redundant or duplicate code? (DRY)
- Do loops have a set length and do they terminate correctly?
- Can debugging or logging code be removed?
- Can any of the code be replaced by library functions?

Security

- If you're using a library, do you check errors it returns?
- Are all data inputs checked?
- Are output values checked and encoded properly?
- Are invalid parameters handled correctly?

Tests

- Do unit tests actually test what they are supposed to?
- Is bounds checking being done?

- Is a test framework and/or library used?

Providing feedback

The main purpose of a code review is to find issues or defects in a piece of code. These issues then need to be communicated back to the developer who proposed the change, so that they can be fixed. Doing this badly can quickly spoil everyone's fun.

Perhaps the most important point in this guide therefore is that the goal of a code review is *not* to provide criticism of a piece of code, or even worse, the person who wrote it. *The goal is to help create an improved version.*

So, when providing feedback, stay positive and constructive. Suggest a better way if possible, rather than just commenting that the current solution is bad. Ideally, submit a patch rather than an issue ticket. And always keep in mind that you're not required to find anything, if the code is fine, it's fine. If it's more than fine, file a compliment!

Most of our projects are hosted on GitHub, so most results will be communicated through pull requests and issues there. However, if you find something particularly bad or weird, consider talking in person, where a lengthy, complicated, or politically sensitive explanation is easier to do.

Communicating results through GitHub

If you are reviewing a pull request on Github, comments should be added in the `Files changed` section, so they can be attached to a particular line of code. Make many small comments this way, rather than a big ball of text with everything in it, so that different issues can be kept separate. Where relevant, refer to existing Issues and documentation.

If you're reviewing existing code rather than changes, it is still handy to use pull requests. If you find an issue that has an obvious fix, you can submit a pull request with a patch in the usual way.

If you don't have a fix, you can add an empty comment to the relevant line, and create a pull request from that as a patch. The relevant line(s) will then light up in the pull request's `Files changed` overview, and you can add your comments there. In this case, either the pull request is never merged (but the comments processed some other way, or not at all), or the extra comments are reverted and replaced by an agreed-upon fix.

In all cases, file many small pull requests, not one big one, as GitHub's support for code reviews is rather limited. Putting too many issues into a single pull request quickly becomes unwieldy.

Reviewing Jupyter notebooks with ReviewNB

For rich diffs and commenting on Jupyter notebooks Github is not very useful, because Github shows differences between Jupyter notebooks as a diff between json files. You can use ReviewNB for reviewing Jupyter notebooks. ReviewNB shows the diff between notebooks in a human readable way. If you configure your esciencecenter.nl email address as your primary email for your Github account here you have access to the academic plan with full benefits.

References

Atwood, Jeff (2006) Code Reviews: Just Do It

Burke, Kevin (2011) Why code review beats testing: evidence from decades of programming research.

McConnell, Steve (2004) Code Complete: A Practical Handbook of Software Construction, Second Edition. Microsoft Press. ISBN-13: 978-0735619678

SMARTBEAR (2016) Best practices for code review.

Release

Releases are a way to mark or point to a particular milestone in software development. This is useful for users and collaborators, e.g. I found a bug running version x. For publications that refer to software, refering to a specific release enhances the reproducability.

Apache foundation describes their release policy.

Release cycles will depend on the project specifics, but in general we encourage quick agile development: *release early and often*

Semantic versioning

Releases are identified by a version number. **Semantic Versioning (semver)** is the most accepted and used way to add numbers to software versions. It is a way of communicating impact of changes in the software on users.

A version number consists of three numbers: major, minor, and patch, separated by a dot: *2.0.0*. After some changes to the code, you would do a new release, and increment the version number. Increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.

Very often package managers depend on `semver` and will not work as expected otherwise.

Releasing code on github

Github makes it easy to do a release straight from your repositories website. See **github releases** for more information.

CHANGELOG.md

A change log is a way to communicate notable changes in a release to the users and contributors. It is typically a text file at the root of your repository called *CHANGELOG.md*. Every release should have relevant entry in change log.

See Keep a CHANGELOG for some best practices.

One command install

To not scare away users and (potential) collaborators, installing the software should be easy, a one command process. The process itself typically includes installing dependencies, compiling, testing, and finally actual installation, and can be quite complex. The use of a proper build system is strongly recommended.

Package in package manager

If your software is useful for a wider audience, create a package that can be installed with a package manager. Package managers can also be used to install dependencies quickly and easily.

- For Python use pip
- For Javascript use npm
- C, C++, Fortran, ... use packages from your distributions official repository. List your actual dependencies in the *INSTALL.md* or *README.md*

Some standard solutions for building (compiling) code are:

- The Autotools: *autoconf*, *automake*, and *libtool*. See the Autotools Documentation, or an introductionary presentation by Thomas Petazzoni
- CMake
- Make

Release quick-scan by other engineer

A check by a fellow engineer to see if the documentation is understandable? can the software be installed? etc.

Think of it as a kind of code review but with focus on mechanics, not code. The reviewer should check if: (i) there is easily visible or findable documentation, (ii) download works, (iii) there are instructions on how to (iv) install and (v) start using software, some of the things in this *scan* could be automated with continuous integration.

Citeable

Create a DOI for each release see Making software citable.

Dissemination

When you have a first stable release, or a subsequent major releases, let the world know! Inform your coordinator and our Communications Advisor so we can write news item on our site, add it to the annual report, etc.

Documentation

Developed programs should be documented at multiple levels, from code comments, through API documentation, to installation and usage documentation. Comments at each level should take into account different target audience, from experienced developers, to end users with no programming skills.

Example of good documentation: A Guide to NumPy/SciPy Documentation

Markdown

Markdown is a lightweight markup language that allows you to create webpages, wikis and user documentation with a minimum of effort. Documentation written in markdown looks exactly like a plain-text document and is perfectly human-readable. In addition, it can also be automatically converted to HTML, latex, pdf, etc. More information about markdown can be found here:

http://daringfireball.net/projects/markdown/

http://en.wikipedia.org/wiki/Markdown

Retext is a markdown aware text editor, that can be used to edit markdown files and convert them into HTML or PDF. It can be found at:

https://github.com/retext-project/retext

Alternatively, 'pandoc' is a command line utility that can convert markdown documents to into several other formats (including latex):

http://johnmacfarlane.net/pandoc/

An Eclipse plugin for previewing the HTML generated by markdown is available on this page:

https://marketplace.eclipse.org/content/markdown-text-editor

Readme

Clear explanation of the goal of the project with pointers to other documentation resources.

Use GitHub flavoured markdown for, e.g., syntax highlighting. (If reStructuredText or another format that GitHub renders is idiomatic in your community, use that instead.) README is targeted towards developers, it is more technical than home page. Keeping basic documentation in README.md can be even useful for lead developer, to track steps and design decisions. Therefore it is convenient to create it from the beginning of the project, when initialising git repository.

- StackOverflow on good readme
- short gist with README.md template
- The art of README from nodejs community

Well defined functionality

Ideally in README.md

Source code documentation

Code comments

Code comments, can be block comments or inline comments. They are used to explain what is the piece of code doing. Those should explain why something is done in the domain language and not programming language - why instead of what.

API documention

API documentation should explain function arguments and outputs, or the object methods. How they are formulated will depend on the language.

Usage documentation

- User manual (as PDF) in the "doc" directory. This is the real manual, targeted at your users. Make sure this is readable by domain experts, and not only software developers. Make sure to include:
  - Netherlands eScience Center logo.
  - Examples.
  - Author name(s).
  - Versions numbers of the software and documentation.
  - References to:
    - The eScience Center web site.
    - The project web site.
    - The Github page of the project.

    - Location of the issue tracker.
    - More information (e.g. research papers).

Documented development setup

(good example is Getting started with khmer development) It should be made available once there is more than one developer working on the codebase. If your development setup is very complicated, please consider providing a Dockerfile and docker image.

Contribution guidelines

Contribution guidelines make it easier for collaborators to contribute, and smooth the process of collaboration.

Guidelines should be made available once the code is available online and there is a process for contributions by other people. Good guidelines will save time of both lead developer and contributor since things have to be explained only once. A good CONTRIBUTING.md file describes at least how to perform the following tasks:

- How to install the dependencies
- How to run (unit) tests
- What code style to use
- Reference to code of conduct
- Mention the git branching model when using anything else than the GitHub flow branching model

An extensive example is Angular.js's CONTRIBUTING.md. Note that GitHub has built in support for a CONTRIBUTING.md file.

Code of conduct

A code of conduct is a set of rules outlining the social norms, religious rules and responsibilities of, and or proper practices for an individual. Such a document is advantagous for collaboration, for several reasons:

- It shows your intent to work together in a positive way with everyone.
- It reminds everyone to communicate in a welcoming and inclusive way.
- It provides a set of guidelines in case of conflict.

contributor covenant

CofC should be attached from the beginning of the project. There is no gain from having it with one developer, but it does not cost anything to include it in the project and will be handy when more developers join.

Documented code style

From the beginning of the project, a decision on the code style has to be made and then should be documented. Not having a documented code style will highly increase the chance of inconsistent style across the codebase, even when only one developer writes code. The Netherlands eScience Center should have a sane suggestion of coding style for each programming language we use. Coding styles are about consistency and making a choice, and not so much about the superiority of one style over the other. A sane set of guides can be found on in google documentation.

How to file a bug report

Describing how to properly report a bug will save a lot of developers's time. It is also useful to point users to good bug report guide like one from Simon Tatham

- An example of such a document for Mozilla projects
- Other example from Ubuntu Docuementation

Explained meaning of issue labels

Once users start submitting issues labels should be documented.

DOI or PID

making your code citable

Identifiers should be associated with releases and should be created together with first release.

Software citation

To get credit for your work, it should be as easy as possible to cite your software.

Your software should contain sufficient information for others to be able to cite your software, such as: authors, title, version, journal article (if there is one) and DOI (as described in the DOI section). It is recommended that this information is contained on a single file.

You can use the Citation File Format to provide this information on a human- and machine-readable format.

Read more in the blog post by Druskat et al..

Print software version

Make it easy to see which version of the software is in use.

- if it's a command line tool: print version on the command line
- if it's a website: print version within the interface
- if the tool generates the output: output file should contain the version of software that generated the output

Use standards

Standard files and protocols should always be a primary choice. Using standards improves the interoperability of your software, thereby improving its usefulness.

Exchange formats

Examples include Unicode W3C, OGN, NetCDF, etc.

Protocols

Examples include HTTP, TCP, TLS, etc.

Contributing to the Guide

Who? You!

This guide is primarily written by the Research Software Engineers at the Netherlands eScience Center. Contributions by anyone (also outside the Center) are most welcome!

Page maintainers

While everybody is encouraged to contribute where they can, we appoint maintainers for specific pages to regularly keep things up to date and think along with contributors. To see who is responsible for which part of the guide see the maintainer listed at the top of a page. If you are interested in becoming a chapter owner for a page that is listed as *unmaintained*, please open a pull request to add your name instead of *unmaintained*.

Editorial board

The editors make sure content is in line with the scope, that it is maintainable and that it is maintained. In practice they will:

- track, lead towards satisfactory conclusion of and when necessary (in case of disagreement) decide on issues, discussions and pull requests,
- flag content that needs to be updated or removed,
- ask for input from page maintainers or other contributors,
- periodically organize sprints to work on content together with everyone interested in contributing; usually in the form of a "Book Dash" together with The Turing Way contributors,

and do any other regular editing tasks.

Currently the team consists of:

- Bouwe Andela @bouweandela (research software engineer)
- Carlos Martínez Ortiz @c-martinez (community manager)

- Patrick Bos @egpbos (technology lead)

Audience

Our eScience Center *RSEs* are the prototypical audience members, in particular those starting out in some unfamiliar area of technology. Some characteristics include:

- They are interested in *intermediate to advanced level* best practices. If there are already ten easily found blog posts about it, it doesn't have to be in the Guide.
- They are a *programmer or researcher* that is already familiar with some other programming language or software-related technology.
- They may be generally interested (in particular topics of eScience practice and research software development in general or how this is done at the eScience Center specifically), but their main aim is towards *practical* application, not to create a literature study of the current landscape of (research) software.

Scope

To make sure the information in this guide stays relevant and up to date it is intentionally low on technical details. The guide contains and links to best practices we use to code and develop research software in our projects.

The main goal: having information available about research software engineering best practices for our colleagues, collaborators and other interested people. It can be information that you can give a colleague starting in some area, for instance, a new language or a new technology.

80% of this goal will be met by the Turing Way. For everything else: we have the Guide.

We focus on eScience Center-specific best practices. These can be generic and complete or specific and highly curated. It depends! For instance, eScience specific content (e.g. we prefer `git` over `svn`) should be in the Guide, while content of interest to a general audience (e.g. it is good practice to use a version control system) should go in The Turing Way. When in doubt, discuss your doubts in an issue.

A few things are excluded:

1. Project related practices (planning, communication, stake holders, management, etc.). These we gather on our intranet pages.
2. Project output is gathered on the Research Software Directory.
3. Generic research software engineering advice that can be added to The Turing Way.

In practice, this means the Guide (for now) will mostly consist of language guides and technology guides.

It can also sometimes function as a staging/draft area for eventually moving content to the Turing Way. However, we will urge you to contribute to the Turing Way directly.

For significant changes / additions, especially new chapters

Please check if your contribution fits in The Turing Way before considering contributing to this guide. Feel free to ask the editors if you are unsure or open an issue to discuss it. If it does not fit, please open an issue to discuss your planned contribution before starting to work on it, to avoid disappointment later.

How?

Style, form

A well written piece of advice should contain the following information:

1. What, e.g. *version control*
2. Why, e.g. *why version control is a good idea*
3. Short how / tl;dr: Recommend one solution for readers who don't want to spend time reading about all possible options, e.g. *at NLeSC we use git with GitHub because...* This is where NLeSC specific info should go if it makes sense to do so.
4. Long how: also explain other options for implementing advice, e.g. *here's a list of some more version control programs and/or services which we can recommend*.

Technical

Please use branches and pull requests to contribute content. If you are not part of the Netherlands eScience Center organization but would still like to contribute please do by submitting a pull request from a fork.

```shell
git clone https://github.com/NLeSC/guide.git
git branch newbranch
git checkout newbranch
```

Add your new awesome feature, fix bugs, make other changes.

To preview changes locally, host the repo with a static file web server:

```shell
python3 -m http.server 4000
```

to view the documentation in a web browser (default address: http://localhost:4000).

To check if there are any broken links use lychee in a Docker container:

```shell
docker run --init -it -v `pwd`:/docs lycheeverse/lychee /docs --config=docs/lychee.toml
```

If everything works as it should, `git add`, `commit` and `push` like normal.

If you have made a significant contribution to the guide, please make sure to add yourself to the `CITATION.cff` file so your name can be included in the list of authors of the guide.

Create a PDF file

We host a PDF version of the guide on Zenodo. To update it a new release needs to be made of the guide. This will trigger a GitHub action to create a new Zenodo version with the PDF file.

Technical details

The basics of how the Guide is implemented.

The Guide is rendered by docsify and hosted on GitHub Pages. Deployment is "automatic" from the main branch, because docsify requires no build step into static HTML pages, but rather generates HTML dynamically from the MarkDown files in the Guide repository. The only configuration that was necessary for this automatic deployment is:

1. The index.html file in the root directory that loads docsify.
2. The empty .nojekyll file, which tells GitHub that we're not dealing with Jekyll here (the GitHub Pages default).
3. Telling GitHub in the Settings -> Pages menu to load the Pages content from the root directory.
4. The _sidebar.md file for the table of contents.

Plugins that we use:

- The docsify full text search plugin
- The docsify Google Analytics plugin

- Prism is used for language highlighting.

If you want to change anything in this part, please discuss in an issue.

Zen of the Guide

1. Help your colleagues.
2. Citing is better than copying.
3. Copying is better than rewriting from scratch.
4. ... but leaving out is often even better.
5. Don't state the obvious.
6. Don't assume that something is obvious.
7. Snippets are friends.
8. Remove outdated content.
9. Better yet, update outdated content.
10. Your practices are just *your* practices. Best practices are shared practices. $N > 1$.
11. Our best practices are just *our* best practices. We don't have to agree with everyone.
12. Best practices are timeless (at least for a year or so).
13. Best practices are never set in stone. They are set in the Guide.
14. Best practices are not always practices.
15. ~~Best practices are not always best practices.~~
16. Kill your darlings.
17. Consider The Turing Way first.
18. Sharing is better than guiding.
19. Guiding is better than turning a blind eye.
20. This Guide shall be under your pillow.