



Netherlands eScience Center

# Software Development Guide

# Table of Contents

- [Introduction](#)
- [Best practices](#)
- [Language Guides](#)
  - [Bash](#)
  - [JavaScript and TypeScript](#)
  - [Python](#)
  - [R](#)
  - [C and C++](#)
  - [Fortran](#)
  - [Rust](#)
- [Technology Guides](#)
  - [GPU programming](#)
  - [UX - User Experience](#)
  - [Datasets](#)
- [Contributing to this Guide](#)
- [Privacy](#)



# Guide

This is a guide to research software development at the Netherlands eScience Center. It is a living document, written by and for our research software engineers (RSEs) and our collaborators.

We write it for two reasons:

1. To have a trusted source for quickly getting started on selected software development topics. We hope this will help RSEs (including our future selves!) to get off to a flying start on new projects in software/technological areas they are not yet familiar with.
2. To discuss and reach consensus on such topics/areas. This in itself is valuable experience! Discussing your practices can be confronting and a bit uncomfortable, but often teaches you new tricks and points of view.

Openness and collaboration are at the heart of the eScience Center, which is why we develop and share these guidelines in the open. [Join us!](#)

## Contents

To get started, check out the checklist of generic research software engineering advice in the [Best Practices](#) chapter. This chapter lists the most important overall attention points while developing research software. For more details, the sections refer to selected resources in community guides that we collaborate with.

If you are looking for more in-depth advice on using a specific programming language, have a look at the [language guides](#). Here we catalogue our experiences with the languages we use the most in our research software development projects. We also provide [technology guides](#) on digital technologies we use often in our projects with research partners.

## Resources

All of the text in this guide is backed by our own experiences in developing high quality research software. However, we also learn from and share knowledge with other community-driven research software guides. The two most important of these are [The Turing Way](#) and the [Research Software Quality Kit](#). Their scope is slightly different, but we collaborate with them when we can.

# Contributing

Please consider contributing to this book! It is a great way to make long-lasting impact by sharing your time-tested knowledge and expertise. You'll hone your writing skills while you're at it.

See the [Contributing to this Guide](#) chapter if you want to know more about how you can help, or ask one of the editors. Currently the editorial team consists of:

- Bouwe Andela [@bouweandela](#) (research software engineer)
- Carlos Martínez Ortiz [@c-martinez](#) (community manager)
- Patrick Bos [@egpbos](#) (technology lead)

# Best Practices for Software Development

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

## Checklists

An easy way to make sure you did not forget anything important is to use a well curated checklist. Great examples can be found via [FAIR Software NL](#). [The Turing Way](#) has specific topical checklists at the end of each of their chapters.

## Version control

Use a version control tool like `git` to track changes in your codebase. This allows you to retrace your steps when debugging, keep your repository clean, easily collaborate with others asynchronously and more. More info: [The Turing Way chapter on Version Control](#), [RSQkit chapter on Version Control](#).

**At the Netherlands eScience Center:** we always use version control and we preferably use GitHub as our online repository and collaboration platform (see the [Project Management Protocol on our intranet](#) (only accessible to Netherlands eScience Center employees)).

## Testing

Tests are important for two reasons: 1. confirming the expected workings of your code while developing for the first time and 2. making sure your features keep working when later on you or others modify the implementation. [The Turing Way gives an overview of the many ways to test code](#).

## Code Reviews

The most effective tool for improving software quality (and sharing knowledge at the same time) is doing code reviews. Have a look at the [The Turing Way chapter on Code Reviewing](#) to learn more about ways to do this.

## 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 audiences, from experienced developers, to end users with no programming skills. In the [Turing Way chapter on Code Documentation](#) you will find a great overview of the how and why of documentation.

## Code Quality

Ways to improve code quality are described in the [Code quality](#) chapter on the Turing Way.

Explore [online tools for software quality improvement](#). Additionally, check our [language guides](#) for language-specific recommendations. [RSQKit: Research Software Quality Kit](#) also has many useful guides including software quality. These guides are result of an international collaboration primarily focusing on research software quality.

## EditorConfig

The eScience Center provides a [shared config file](#) for IDEs and text editors. This file helps standardize coding styles across projects.

## Namespaces

If your programming language supports namespaces, use your organization or project-specific namespace.

**At the Netherlands eScience Center:**, the recommended namespace is `nl.esciencecenter`, or adapt it to a namespace that aligns with your project's context.

## Use standards

Standard files and protocols should always be a primary choice. Using standards improves the interoperability of your software, thereby improving its usefulness. Examples include exchange formats like Unicode, NetCDF, and W3C web standards, and protocols like HTTP, TCP, TLS.

## Licensing

Since source code is protected by copyright, to allow people to use your code it needs a license. For more information, see [The Turing Way chapter on licensing](#) or the [RSQkit Licensing software task](#).

**At the Netherlands eScience Center:** our first choice is the Apache v2 license. See the [Project Management Protocol on our intranet](#) (only accessible to Netherlands eScience Center employees) for more details on licensing and our intellectual property policies.

## Software management plans

The Netherlands eScience Center and [NWO](#) have authored the [practical guide to software management plans](#) ([see also](#)). For our projects we recommend using [our Software Sustainability Protocol](#), which is based on these guidelines. For more information you can also [read here](#).

## Releases

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, referring to a specific release enhances the reproducibility. See [the RSQkit task on Creating code releases](#) for the most essential guidelines. The Turing Way offers many related tips in their [chapter on Making Research Objects Citable](#), like how to make code citable with CITATION.CFF files.

## Packaging

A related, but separate topic is packaging, which allows users to conveniently install your released software. Most [languages](#) and OS'es have their particular ways of doing this. The Turing Way offers advice on [making reproducible environments](#), in which packaging is an essential component.

## Know your tools

In addition to the advice on the best practices above, 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 editor with extensive plugin ecosystem that can make it as powerful as most IDEs
- [JetBrains IDEs](#) - specialized IDEs for Python, C++, Java and web, all using the IntelliJ framework
- [Eclipse](#) - a bit older but still nice

Text editors:

- [vim](#) - classic text editor
- [emacs](#) - classic text editor



# Language Guides

*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 recommended 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 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

```
set -euo pipefail
```

bash

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

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

bash

## 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

# JavaScript

Page maintainer: Ewan Cahen [@ewan-esience](#)

[JavaScript](#) (JS) is a programming language that is one of the three (together with [HTML](#) and [CSS](#)) core technologies of the web. It is essential if you want to write interactive webpages or web applications, because JavaScript is, apart from [WebAssembly](#), the only programming language that runs in modern browsers. Furthermore, JS can also run [outside of the browser](#), e.g. for running short scripts or full-blown servers.

## Getting started

A good introductory tutorial on JavaScript is [this one from W3Schools](#).

Another source of information for JavaScript (and web development in general) is the [MDN Web Docs](#).

## Frameworks

Many people will jump straight to using a framework when building a web application. We, however, recommend that you learn the fundamentals first and get an impression of what problems frameworks are trying to solve for you. Read, for example, this article on [how the web works](#) a look at this [introduction to the DOM](#).

A good video summary on the history of frameworks and the problems they try to solve can be found [here](#).

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](#)

## React

[React](#) is a framework which can be 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 an 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 an 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 it 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. Read [this article from W3Schools](#) for more info.

There is documentation for each browser on their [dev tools](#):

- [Firefox](#)
- [Chrome](#)
- [Edge](#)
- [Safari](#)

There are also debugging guides for the various JS runtimes:

- [Node.js](#)
- [Deno](#)
- [Bun](#)

When using a (meta-)framework, also have a look at its documentation.

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/uglifyfication/transpiling tools.

## Hosting data files

To display web pages (HTML files) with JavaScript, you can't use any file system URL due to safety restrictions. You should use a [web server](#) (which may still serve files that are local). A simple web server can be started from the directory you want to host files with:

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

bash

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

# Documentation

[JSDoc](#) (similar to [JavaDoc](#)), parses your JavaScript files and automatically generates HTML documentation, based on the JSDoc comments you put in the code.

## Testing

The various runtimes have testing functionality included, so you don't have to install extra dependencies:

- [Node.js](#)
- [Deno](#)
- [Bun](#)

If these don't suffice, a nice overview of popular testing frameworks can be found [here](#).

## Testing with browsers

To interact with web browsers use [Selenium](#).

## Coding style

### Formatters

A formatter is a tool to make your source code look consistent and easy to look at. In web development, the most used formatter is [Prettier](#), which can [integrate with many editors](#). You could [set up a GitHub action](#) that rejects pull requests that are not formatted properly.

When using Deno, you can also use its [built-in formatter](#).

An alternative to Prettier is [Biome](#), which also includes a linter.

In any case, remember to use tabs for indentation for the [purpose of accessibility](#).

### Linters

A linter is a tool to check your code quality, in order to prevent bugs. The most used linter is [ESLint](#). It has [many integrations](#)

When using Deno, you can also use its [built-in linter](#).

An alternative to ESLint is [Biome](#), which also includes a formatter.

Also have a look at the [Airbnb JavaScript Style Guide](#) or the W3Schools page on [JavaScript best practices](#).

## Code quality analysis tools and services

For more in-depth analyses, you can use a code quality and analysis tool.

- [SonarCloud](#) is an open platform to manage code quality which can also show code coverage and count test results over time. It easily [integrates with GitHub](#).
- [Codacy](#) can analyze [many different languages](#) using open source tools. It also offers [GitHub integration](#).
- [Code climate](#) can analyze JavaScript (and Ruby, PHP). Can analyze Java (best supported), C, C++, Python, JavaScript and TypeScript.

## 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.

## 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 without significant differences, 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, check out the [official documentation](#). Note that Deno and Bun support TypeScript [out of the box](#).

## 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:

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

shell

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

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

shell

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

Install it with:

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

shell

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. For more info, see the [Javascript section on debugging](#)

## Documentation

Just like [JSDoc](#) for JavaScript, [TypeDoc](#) can automatically generate HTML documentation for your code.



# 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, and for many other types of projects: workflow management, visualization, natural language processing, 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.

- [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 up the 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 the [Python 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`.

## Type hints

Since [PEP 484](#), which was first implemented in Python 3.5 (released in 2015), Python has gained the ability to add type information to variables. These are not types, as in typed languages; they are *hints*. Naively, one could say they are a new type of documentation. However, in practice they are far more than this, because they do have their own special syntax rules and are thus parsable. In fact, some tools have started to make use of this in runtime modules as well, making them more than hints for tools like Pydantic, FastAPI and Typer (all described below). See [this guide](#) to learn more about type hints.

Some tools to know about that make use of type hints:

- [Type checkers](#) are static code analysis tools that check your code based on the type hints you provide. It is highly recommended that you use a type checker. Choose [mypy](#) if you are unsure which one to choose.
- Tools to build documentation from source code have extensions that can show type hints in the generated documentation to make your code easier to understand. Popular examples are [sphinx autodoc](#), [sphinx autapi](#), and [mkdocstrings](#).
- [Pydantic](#) is a widely used data validation library that allows you to automatically validate instances of dataclasses at runtime. This means that for this tool the type hints are no longer just hints or a form of documentation, but have actual effects. Essentially, a fully Pydantic-enriched application (in "strict mode") is like having Mypy at runtime (there is also a "tolerant" mode that lets some common types slip through without errors). It effectively turns Python into a statically typed language.
- Most editors nowadays make use of type hints for autocompletion. If the editor knows the type of your variable, for instance, it can autocomplete attributes or methods of that class.

We recommend using type hints, where possible and *practical*. Type hints are still being actively developed; not everything one would like to be able to express in a compact way can yet be achieved. This is why, for instance, [NumPy](#) arrays and machine learning library (e.g. [Pytorch](#), [Tensorflow](#)) "tensor" types still (in 2024) have awkward type hinting. Crucial information that one would typically want to encode for array type input arguments are shapes, but this is not yet possible. Other important libraries, like [Matplotlib](#), have very complex functions that take in many possible types of arguments, leading to overly complex variable types. Such huge types clutter your code tremendously, so they are not typically encouraged.

# 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 `mock` 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 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.
- `puddb` 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/spisode/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](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](#)

## Documentation

It is recommended that you [write documentation](#) for your projects and publish it on an interactive webpage. A popular and recommended solution for hosting documentation is [Read the Docs](#). It can automatically build documentation for projects hosted on [GitHub, GitLab, and Bitbucket](#).

## Building documentation

There are several tools for building webpages with documentation. At the eScience Center, we mostly use [Sphinx](#) (more established) and [MkDocs](#) (newer).

User guides and other text documents are typically written in [Markdown](#) or [reStructuredText](#). Sphinx supports both formats, while MkDocs only supports Markdown. Markdown has the advantage that it's easier to read for humans so it may be easier to work with and contribute to. reStructuredText is easier to read for computers so may be more suitable for complex projects.

Python uses [Docstrings](#) for code documentation. You can read a detailed description of docstring usage in [PEP 257](#). Both Sphinx and MkDocs can generate documentation webpages from docstrings. There are two popular Sphinx extensions for generating documentation: [autoapi](#) (newer and more lightweight) and [autodoc](#) (more established). For MkDocs the [mkdocstrings](#) package is available. We recommend using the [NumPy documentation style](#), as that is widely used in the scientific Python ecosystem.

You can also integrate entire Jupyter notebooks into your documentation with [nbsphinx](#) or [mkdocs-jupyter](#). This way, your demo notebooks, for instance, can double as documentation. Of course, the notebooks will not be interactive in the compiled webpage, but they will include all code and output cells and you can easily link to an interactive version from the compiled documentation.

It is recommended that you [routinely test any code examples in your documentation](#).

## 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 environments. 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 processes.
- 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)
- [FastAPI](#): again, similar to flask in functionality, but uses modern Python features like async and type hints with runtime behavioral effects.

We have recommended `flask` in the past, but FastAPI has become more popular recently.

## 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`.
- `Typer`: make a command-line application by using type hints with runtime effects. Very low on boilerplate for simple cases, but also allows for more complex cases. Uses `click` internally.

# R

Page maintainers: [Malte Lüken](#) and [Pablo Rodríguez-Sánchez](#) .

## What is R?

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 to that of Matlab and Python in terms of compactness and readability, which makes it a good prototyping language for 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](#).

## Some crucial differences with Python

Are you familiar with Python? Then kickstart your R journey by reading this [blog post](#).

## Recommended sources of information

All R functions come with documentation in a standardized format. Some R packages have their own google group. Further, stackoverflow and standard search engines can lead you to answers to issues.

If you prefer books, consider the following resources:

- [R for Data Science](#) by Hadley Wickham,
- [Advanced R](#) by Hadley Wickham,
- [Writing better R code](#) by Laurent Gatto.

## 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()`.

Said this, it is highly recommended to use an integrated development environment (IDE). The most popular one is [RStudio / Posit](#). It is free and quite powerful. It features editor with code completion, command line environment, file manager, package manager and history lookup among others.

It comes with many menus and key bindings (visible when you hover your mouse over the menu item). For instance, you can run code sections by selecting them and pressing `Ctrl+Enter`.

Note 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

We recommend following the [Tidyverse style guide](#). Its guidelines can be automatically followed using linters such as:

- [styler](#)
- [lintr](#)

## The `<-` operator

Assigning variables with `<-` instead of `=` is recommended, although **most** of the time both are equivalent.

If you are interested in the controversy around assignment operators, check out this [blog post](#).

## %>% and |>

The symbols `%>%` and `|>` represent the pipe operator. The first one is part of the `magrittr` package, and it gained so much popularity that a similar operator, `|>`, was added as part of native R since version 4.1.0. For details on the differences between the two, see this [blog post](#). They just add syntactic sugar to the way we pass a variable to a function. The example below shows its basic behavior:

```
var %>% function(params)
# Is equivalent to
function(var, params)
```

These operators are pretty useful for composing functions, and very often appear concatenated:

```
grades |> remove_nans() |> mean() |> print()
```

You can think of it as a production chain, where an object (the `grades`) passes through three machines, one that removes the `NaN`s, another one that takes the mean, and a last one that prints the result.

## Recommended additional packages and libraries

One of the strengths of R is its community, that creates and maintains a constellation of packages. Very rarely will you use just base R. Here we give you a list of usual packages, starting by one solving the first problem you'll find... how to manage that many packages!

### Managing environments with `renv`

`renv` allows you to create and manage a dependencies library on a per-project basis. It also keeps track of the specific versions of each package used in the project, which is great for reproducibility... and avoiding future headaches!

### Plotting with basic functions and `ggplot2` and `ggvis`



For a generic impression about plotting with R, 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 embedded 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

Speeding up code always start with knowing where your bottlenecks are. The following profiling tools will help you doing so:

- Introduction to [profiling in R](#)

Some rules of thumb that can quickly improve your code are the follwing:

- Avoid loops, use `apply` functionals instead
- Try to use vectorized functions
- Checkout the `purrr` package
- If you are really in a hurry, consider communicating with `C++` code using `Rcpp` .

For a deeper introduction to the many optimization methods, check the free ebook:

- [Efficient R programming](#), by Colin Gillespie and Robin Lovelace.

## 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](#). For a quicker introduction, consider this software Carpentries' [lesson on R packages](#), originated and developed at our Center!

### 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.

Function level comments starting with `#'` are used by `roxygen` to automatically generate the `.Rd` files. This means that you **don't have to edit the `.Rd` files directly.**

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. Read more about `roxygen` syntax on it's [github page](#). `roxygen` will also populate `NAMESPACE` file which is necessary to manage package level imports.

## Available templates

Most of the templating is natively managed by the `usethis` package. It contains functions that create the boilerplate for you, reducing the burden on your memory and reducing chances for errors. In the snippet below you can see how it feels to use it.

```
usethis::create_package()      # Creates a package structure
usethis::use_readme_md()      # Adds a readme
usethis::use_apache_license() # Adds an Apache License
usethis::use_testthat()       # Adds the testing infrastructure
usethis::use_citation()       # Adds a citation file
# etc...
```

Having said this, these others can serve as inspiration:

- <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. We recommend using GitHub actions.

### 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 program, 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 allows 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 important 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 hesitant 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](#): 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 best practices](#).
- [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.

# Rust

Page maintainer: [Rodrigo V. Honorato](#)

Rust is a modern programming language designed to provide both high performance while enforcing memory safety through its unique ownership system and borrow checker. Developed by Mozilla and first released in 2015, Rust has rapidly gained popularity for its ability to prevent common programming errors at compile time. It is commonly categorized as a systems programming language but over the last few years its ecosystem has grown considerably and Rust is being adopted as a general programming language.

Rust is increasingly adopted in **research software** for its unique blend of speed, safety, and modern tooling. It powers everything from high-throughput DNA sequencing pipelines to climate simulations, where even minor memory errors could invalidate results. By eliminating entire classes of bugs (e.g., null pointers, race conditions, type mismatches), Rust lets researchers focus on science, not on debugging.

It is however a **low-level** language, which gives you direct control over hardware and memory (like [C/C++](#)). For comparison, [Python](#) is a **high-level** language that prioritizes readability by abstracting these details - in Python you don't ever need to think about allocating or freeing memory as the interpreter takes care of it, making the code slower but much easier to program. In a **low-level** language you need to manage it yourself. Because Rust runs "closer to the metal", it achieves blazing-fast performance - similar to [C/C++](#) while avoiding common memory-safety and concurrency bugs.

Here are some of Rust's key characteristics:

- **Memory Safety:** Rust's unique ownership system guarantees memory safety at compile time, eliminating crashes from null pointers, dangling references, or leaks.
- **Type Safety:** Strict compile-time checks ensure variables, data types, and operations are error-free, so there will be no surprises at runtime.
- **Zero-Cost Abstractions:** High-level syntax (e.g., iterators, traits) compiles to machine code as efficiently as hand-written low-level code.
- **Fearless Concurrency:** Built-in rules prevent data races, letting you write safe, parallel code without runtime crashes.
- **Expressive Enums & Pattern Matching:** Enums can hold data, and match ensures all cases are handled—no forgotten edge cases.
- **Traits for Polymorphism:** Define shared behavior across types without runtime overhead.

- **Rich Ecosystem:** Tools like [Cargo](#) (package manager), [Clippy](#) (linting), [crates.io](#) (libraries) and [rustdoc](#) (documentation) streamline development.

rust

```
// Ownership in action: the compiler tracks who "owns" data.
fn main() {
    // Lets declare a string, here `s` owns it
    let s = String::from("hello");

    // Borrow `s` as a read-only reference (no ownership transference)
    let len = calculate_length(&s);

    // `s` still owns the data and we can use it
    println!("{}", s, len);
}

fn calculate_length(s: &str) -> usize {
    s.len()
}
```

## Getting started

To get started you will first need to install Rust, this can be done via [rustup](#) which is a command line tool for managing Rust versions and tools.

On Linux/MacOs:

bash

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

On Windows, [see the instructions here](#).

Cargo is Rust's build system and package manager and is installed by [rustup](#) . You can use it to create a project:

bash

```
cargo new rust_project
```

This will create the project folder structure, add a `Cargo.toml` and a `src/main.rs` which contains a placeholder "Hello world", so you can already build this `rust_project`

bash

```
cd rust_project
cargo build --release # using --release will build the optimized binary
./target/release/rust_project # execute the binary
```

## Learning

Its unique approach to memory management (ownership, borrowing and lifetimes) and the strict compiler can feel daunting at first - especially if you are accustomed to high-level languages like [python](#) or [javascript](#). Learning Rust can be challenging as some new concepts, such as the borrow checker, may take time to be internalized.

Keep in mind that in the long run all the effort pays off. The code produced will be faster while having *fewer bugs* (thanks to the opinionated compiler), you will learn *transferable skills* that will make you a better programmer in other languages. The general mindset should be **start small and embrace the compiler**.

To learn it, you only need:

- [The Rust Book](#): This is the official book and it is very well written and easy to follow. It contains all the information you need to gain a deep understanding of Rust. It contains a fully guided tutorial on how to write a Guessing game as your first project.
- [Rust by Example](#): This contains smaller examples of how to use the language, and it is a good complement to the book or when you need to quickly look up how to do something.
- [Rustlings](#): Fully interactive exercises that will help you get used to the syntax and the concepts of the language - it is paired with the book, so you should be doing the exercises as you go through the book.
- [Rust Playground](#): Lets you experiment with Rust online in your browser



# Technology Guides

*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. [Wikipedia](#)

## Table of content

- UX disciplines
- Design thinking process
- Designing software
- Tools and Resources

## UX disciplines

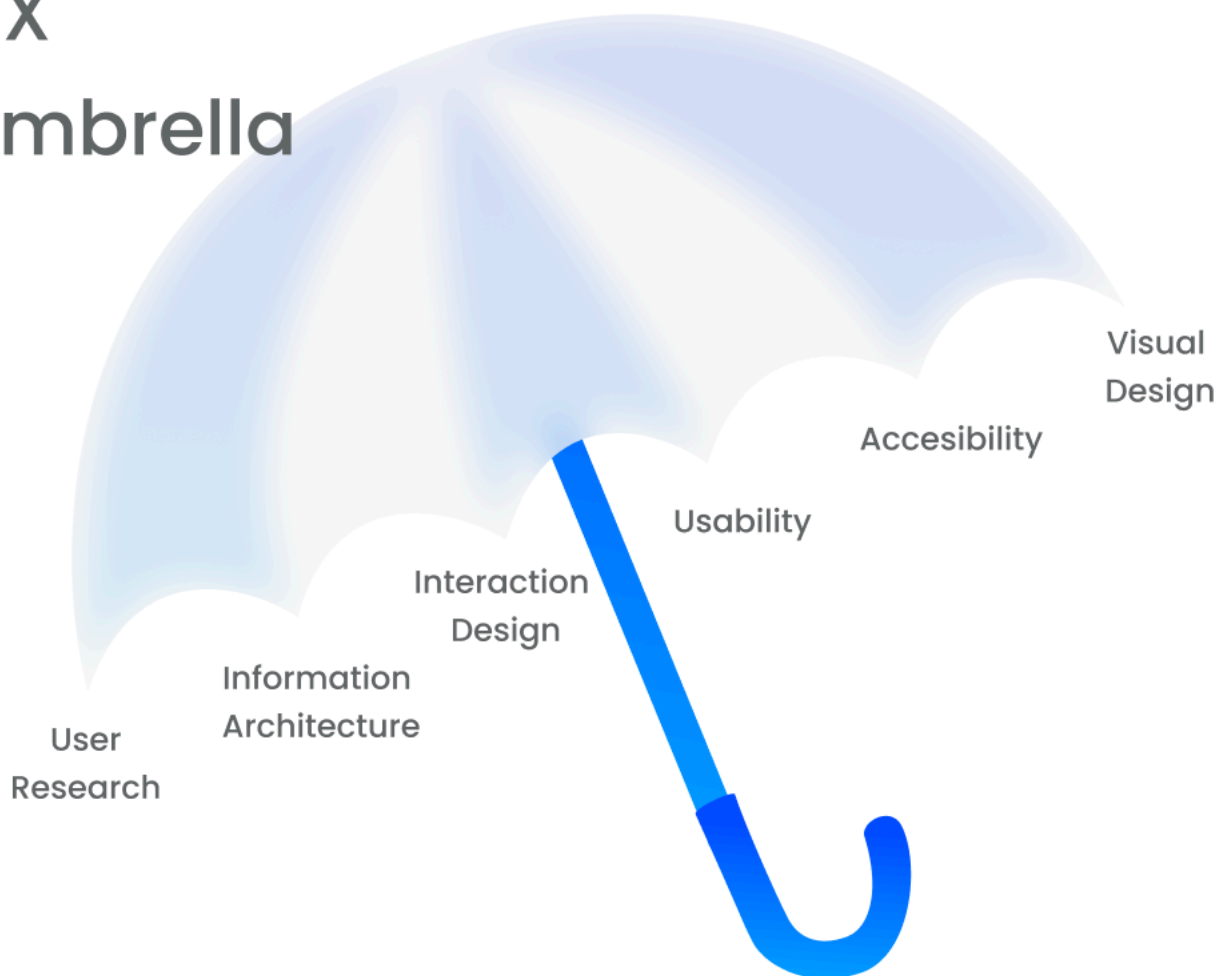
The principles and indications taught by [interaction-design.org](#) can be useful in the process of creating research software.

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:

# UX umbrella



J.G.Gonzalez @esciencecenter

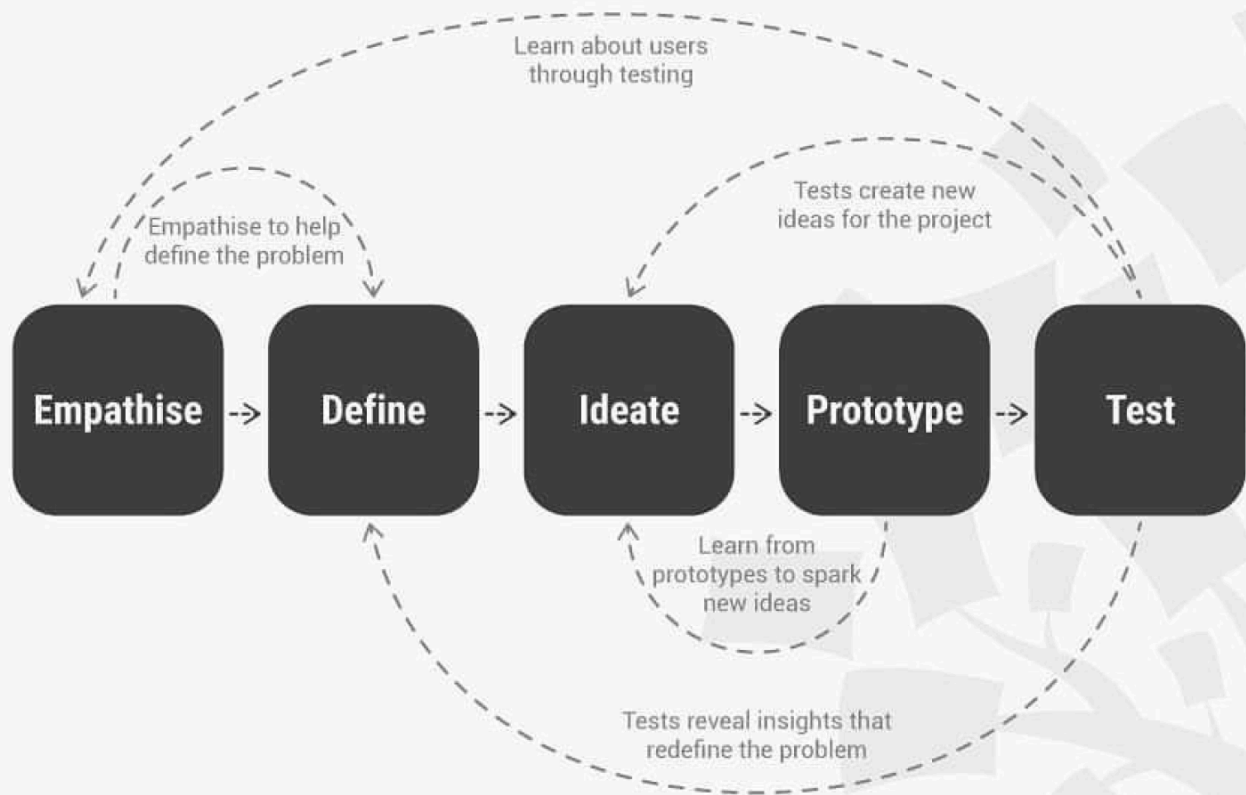
*Author/Copyright holder: J.G. Gonzalez and The Netherlands eScience Center. Copyright: Apache License 2.0*

## 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



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

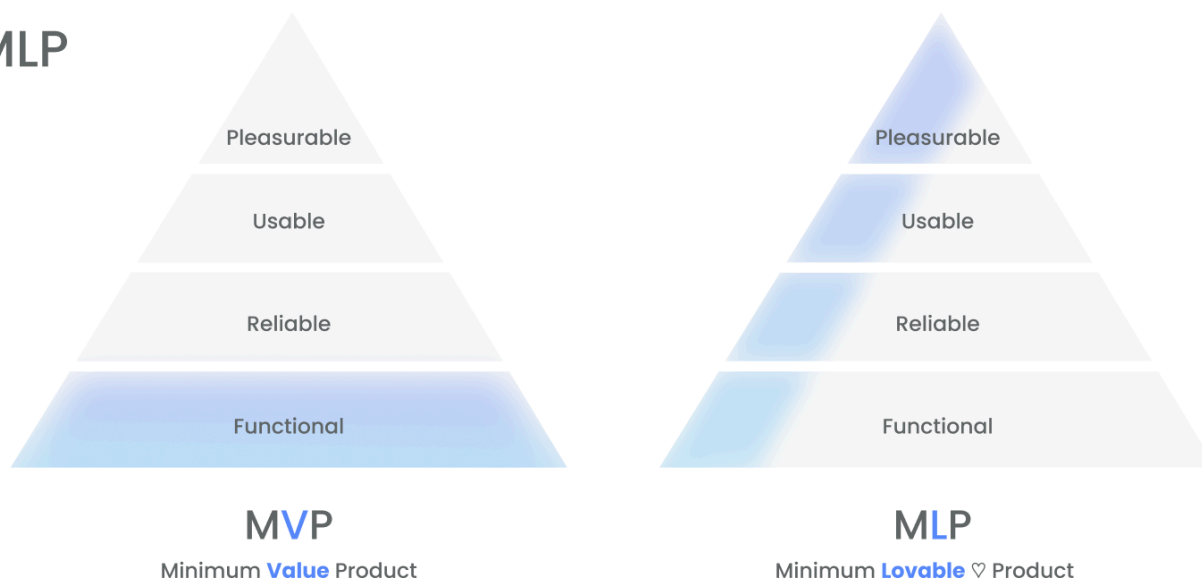
Heuristics, or commonly known 'as the rule of thumb,' play a significant role when users interact with software. The Nielsen/Norman group has a top [10 Usability Heuristics for User Interface Design](#) to consider when developing software.

## 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.

## MVP VS MLP



J.G.Gonzalez @esciencecenter

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

While MVP (Minimun 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 operating system, or shell from preemptively killing it. You can choose a value about 50% of your system's RAM.

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

sql

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

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

sql

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



# Contributing to this Guide

- [Who? You!](#)
- [Audience](#)
- [Scope](#)
- [How?](#)
- [Technical details \(docsify\)](#)
- [Zen of 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
cd guide
git branch newbranch
git checkout newbranch
```

Please install [pre-commit](#) and enable the pre-commit hooks by running

```
pre-commit install
```

to automatically format your changes when committing.

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

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

```
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:

```
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.

# Privacy policy

We collect anonymised user data that helps us to monitor the effectiveness of our website. No personally identifiable information is recorded and no cookies containing such information are set in your browser session.