Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 03

```julia
using PlutoUI
```

Julia workflows

When working with Julia, we can choose between a number of workflows.

Pluto notebook

This ist what you see in action here. After calling pluto, you can start with an empty notebook and add cells.

Great for teaching or trying out ideas.

Jupyter notebook

With the help of the package **IJulia.jl** it is possible to work with Jupyter notebooks in the browser. The Jupyter system is rather complex and Pluto hopefully will be able to replace it, in particular because of its reactivity features.

"Classical" workflow

Use a classical code editor (emacs, vi or whatever you prefer) in a separate window and edit files, when saved to disk run code in a console window.

With Julia, this workflow has the disadvantage that everytime Julia is started, the just-in-time compiler (JIT) needs to recompile all the code to be run if its compiled version is not cached.

While a significant part of the compiled code of installed packages is cached, Julia needs to assume that the code you write can have changed.

- Remedy: Never leave Julia, start a permanent Julia session, include edited code after each change.

The REPL

aka **R**ead - **E**val - **P**rint - **L**oop or Julia command prompt

One enters the REPL when one starts julia in a console window without giving filename.

The REPL allows to execute Julia statements in an interactive fashion. It has convenient editing capabilities.

Helpful commands in the REPL default mode:

commmand	action
`quit()` or `Ctrl+D`	exit Julia
`Ctrl+C`	interrupt execution
`Ctrl+L`	clear screen
Append `;`	suppress displaying return value
`include("filename.jl")`	source a Julia code file and execute content

The REPL has different modes which can be invoked by certain characters:

mode	prompt	enter/exit
`Default`	`julia`	`backspace in other modes to enter`
`Help`	`help?>`	`? to enter`
		`type command name to search`
`Shell`	`shell>`	`; to enter`
		`type command to execute`
`Package manager`	`Pkg>`	`] to enter`

Revise.jl

The **Revise.jl** package allows to keep track of changed files used in a Julia session if they have been included via `includet` (`t` for "tracked"). It controls recompilation of changed code - only those parts which indeed have changed are newly compiled by the JIT, moreover, recompilation is triggered automatically, no need to include code again and again.

In order to make this work, one needs to add

```
if isinteractive()
    try
        @eval using Revise
        Revise.async_steal_repl_backend()
    catch err
        @warn "Could not load Revise."
    end
end
```

to the startup file `~/.julia/config/startup.jl` and to run Julia via `julia -i`.

`Revise.jl` also keeps track of packages loaded and their changes. In this setting it also can be used with Pluto.

Recommendation

When using the REPL based workflow, don't miss Revise.jl and try to find a Julia mode for your favorite editor which provides auto-indentation, highlighteing etc. Mine (emacs) has one.

IDE based workflow

Use an IDE (integrated development environment). Currently the best one for Julia is **Visual Studio Code** with the **Julia extension**.

For introductory material, see the tutorial information given upon starting of a newly installed instance of `code`. For the Julia extension, find videos on `code` **Julia for Talented Amateurs**.

Packages

Structure of a package

- Packages are modules searched for in a number of standard places and as git repositories on the internet
- Locally, each package is stored in directory named e.g. `MyPack` for a package `MyPack.jl`.
- Structure of a package Directory:
  - Subdirectory `MyPack/src` for sources
  - Main source `MyPack/src/MyPack.jl` defining a module named `MyPack`
  - Further Julia sources in `MyPack/src/` included by `MyPack/src/MyPack.jl`
  - Code for unit testing in `MyPack/test`
    - a well designed package has a good number of tests which are run upon every upload on github
  - Code for documentation generation in `MyPack/docs`
    - a well designed package has documentation generated upon every upload on github
  - License
    - Packages in the general registry (see below) are required to have an open source license
  - Metadata

Metadata

Package metadata are stored in `MyPack/Project.toml`

- Name
- Unique Universal Identifier (UUID) - a long character string hopefully unique in the world
- Author
- Version number
- Package dependencies (names and UUIDs)
- **Version bounds** for package dependencies

Environments

Environments (projects) are essentially lists of packages used in a current Julia session. An environment is described by a directory containing at least two metadata files:

- `Project.toml` describing the list of packages required for the project
- `Manifest.toml` describing the actually installed versions of the required packages and *all their dependencies*

Global environment

By default, a global environment stored in `.julia/environments/vX.Y` under the user home directory will be used

Local environments

In oder to avoid version clashes for different projects, one can activate any directory - e.g. `mydir` as a local package environment by invoking Julia with

```
julia --project=mydir
```

Package manager

- Default packages (e.g. the package manager Pkg) are always found in the `.julia` subdirectory of your home directory
- The package manager allows to add packages to your installation by finding their git repositories via the **Julia General Registry** or another registry
  - Packages are found via the UUID
  - During package installation, compatibility is checked accordint to the `[compat]` entries in the respective `Project.toml` files

Basic package manager commands

The package manager can be used in two ways: via the Pkg REPL mode or via Julia function calls after havig invoked `using Pkg`.

Function	`pkg` mode	Explanation
`Pkg.add("MyPack")`	`pkg> add MyPack`	`add MyPack.jl to current environment`
`Pkg.rm("MyPack")`	`pkg> rm MyPack`	`remove MyPack.jl from current environment`
`Pkg.update()`	`pkg> up`	`update packages in current environment`
`Pkg.activate("mydir")`	`pkg> activate mydir`	`activate directory as current environment`
`Pkg.instantiate()`	`pkg> instatiate`	`populate current environment according to Manifest.toml`
`Pkg.test("MyPack")`	`pkg> test mypack`	`run tests of MyPack.jl`
`Pkg.status()`	`pkg> status`	`list packages`

For more information, see the **documentation of the package manager**

Package management in Pluto notebooks

- Pluto (version >=0.16) contains an "automatic" package manager on top of `Pkg`
- Every Pluto notebook contains `Project.toml` and `Manifest.toml` and activates its own environment upon start
- All package versions for a Pluto notebook are fixed $\Rightarrow$ Reproducibility

FAIRness

The **FAIR principles** are fundamental for the role of data in modern research based on good scientific practice. The almost exactly can be applied to software as well - software can be seen as a kind of data.

Findability

"The first step in (re)using data is to find them. Metadata and data should be easy to find for both humans and computers. Machine-readable metadata are essential for automatic discovery..."

- Package metadata
- General registry

Accessibility

"Once the user finds the required data, she/he/they need to know how can they be accessed, possibly including authentication and authorisation"

- Published packages available (mostly) via `github`

Interoperability

"The data usually need to be integrated with other data. In addition, the data need to interoperate with applications or workflows for analysis, storage, and processing"

- Julia supports composability of packages based on interface oriented design of data structures - we will cover this later in the course

Reproducibility

"The ultimate goal of FAIR is to optimise the reuse of data. To achieve this, metadata and data should be well-described so that they can be replicated and/or combined in different settings."

- `Manifest.toml` metadata files are created with reproducibility in mind
- Package version bounds ensure composability across compatible versions of Julia packages - allowing to prevent updates with breaking changes

"Side effects"

- fast pace of development of independent and interoperable packages
- possibility to create up-to-date documentation
- culture of bug fixing via issues and pull-requests to other packages