```
using Pkg; Pkg.activate(mktempdir()); Pkg.add("PlutoUI"); using PlutoUI
```

Code structuring and interaction with other languages

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.

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 very complex and Pluto hopefully will be able to replace it.

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 JIT needs to recompile the packages involved. So the idea is to not leave Julia, but to start a permanent Julia session, and include the code after each change.

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"). In orde to make this work, one should 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. It also can be used with Pluto.

Modern workflow

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

Structuring code: modules, files and packages

- Complex code is split up into several files which can be included
- Need to avoid name clashes for code from different places

Modules

Modules allow to encapsulate implementation into different namespaces

```
Main.workspace3.TestModule
```

```
module TestModule
  function mtest(x)
      return "mtest: x=$(x)"
  end
  export mtest
end
```

```
"mtest: x=2"
```

```
TestModule.mtest(2)
```

Packages

- Packages are modules searched for in a number of standard places
- Each package is a directory named `Package` with a subdirectory `src`
- The file `Package/src/Package.jl` defines a module named `Package`
- More structures in a package:
  - Documentation resources
  - Test code
  - Metadada: Dependency description, UUID (Universal unique identifier)...
- 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 by finding them via the registry and downloading them.

```
String["AbstractTrees", "Adapt", "ArgCheck", "ArgParse", "ArnoldiMethod", "ArrayInter
```

```
readdir("/home/fuhrmann/.julia/packages/")
```

```
String[".github", ".gitignore", ".travis.yml", "LICENSE.md", "NEWS.md", "Project.toml
```

```
readdir("/home/fuhrmann/.julia/packages/AbstractTrees/vOsoQ/")
```

- After importing a package via the `import` statement, all functions from a package are available via their name prefixed with the name of the package.
- The `using` statement makes these names available without prefix.

Calling code from other languages

C

- C language code has a well defined binary interface
  - `int` ↔ `Int32`
  - `float` ↔ `Float32`
  - `double` ↔ `Float64`
  - C arrays as pointers
- Create a C source file:

```
cadd_source = "double cadd(double x, double y) \n{ \n   return x+y; \n}\n"
```

```
cadd_source="""
double cadd(double x, double y)
{
   return x+y;
}
"""
```

```
open("cadd.c", "w") do io
    write(io,cadd_source)
end;
```

Compile to a shared object (aka "dll" on windows) using the gcc compiler:

```
Process(`gcc --shared cadd.c -o libcadd.so`, ProcessExited(0))
```

```
run(`gcc --shared  cadd.c -o libcadd.so`)
```

- Define wrapper function `cadd` using the Julia `ccall` method
  - `(:cadd, "libcadd")`: call cadd from `libcadd.so`
  - First `Float64`: return type
  - Tuple `(Float64,Float64,)`: parameter types
  - `x,y`: actual data passed
- At its first call it will load `libcadd.so` into Julia
- Direct call of compiled C function `cadd()`, no intermediate wrapper code

```
cadd (generic function with 1 method)
```

```
cadd(x,y)=ccall((:cadd, "libcadd"), Float64, (Float64,Float64,),x,y)
```

```
4.0
```

```
cadd(1.5,2.5)
```

- Julia and many of its packages use this method to access a number of highly optimized linear algebra and other libraries

Python

- Both Julia and Python are homoiconic language, featuring *reflection*
- They can parse the elements of their own data structures ⇒ possibility to automatically build proxies for python objects in Julia

The PyCall package provides the corresponding interface:

```
Pkg.add("PyCall"); using PyCall
```

Create a python source file:

```
pyadd_source = "def add(x,y):\n    return x+y\n"
```

```
pyadd_source="""
def add(x,y):
    return x+y
"""
```

```
open("pyadd.py", "w") do io
    write(io,pyadd_source)
end;
```

```
pyadd =
PyObject <module 'pyadd' from '/home/fuhrmann/Wias/teach/scicomp/scicomp/pluto/pyadd.py'>
```

```
pyadd=pyimport("pyadd")
```

```
9.7
```

```
pyadd.add(3.5,6.2)
```

- Julia allows to call almost any python package
- E.g. matplotlib graphics - this is the python package behind PyPlot (there are more graphics options in Julia)
- There is also a **pyjulia** package allowing to call Julia from python

Other languages

- There are ways to interact with C++, R and other langugas
- Interaction with Fortran via `ccall`