Notes 4: Managing Julia

Chris Paciorek

2025-01-30

Table of contents

Introduction

This document is the fourth of a set of notes, this document focusing on managing and interacting with Julia, including working with and creating Julia packages. The notes are not meant to be particularly complete in terms of useful functions (Google and LLMs can now provide that quite well), but rather to introduce the language and consider key programming concepts in the context of Julia.

Given that, the document heavily relies on demos, with interpretation in some cases left to the reader.

Note that for many of the package/project-related demos, I haven't run the code as part of the document rendering as the output is large and it affects that state of Julia in ways that are not helpful.

> **Warning**
>
> There's a lot of configuration details in the section on using packages. It's not critical to absorb it all at this point, but it's here for when you need it. In part I went into detail as I was trying to understand the details myself.

Style guide

Some highlights from the Julia style guide include:

- 4 spaces per indentation level
- Append `!` to names of functions that modify inputs. Mutated input(s) should appear first in the arguments.
- Type and module names: use capitalized/UpperCamelCase.
- Function names: use lower-case squashed names (with underscores/snake_case as needed) (e.g., `isequal` or possibly `is_equal`).
- Use `isa` and `<:` rather than `==` for type comparisons.
- For hard-coded numbers, use `Int`s rather than `Float`s for integer-valued numbers, e.g., `2 * x` rather than `2.0 * x`. (The latter will unnecessarily force conversion to `Float` when `x` is an `Int`.)

File operations (I/O)

Here are some basic examples of reading data from files.

```
filename = joinpath("..", "data", "cpds.csv")
lines = readlines(filename)

using CSV, DataFrames

data = CSV.read(filename, DataFrame);
typeof(data.year)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

```
typeof(data.country)
```

```
PooledVector{String15, UInt32, Vector{UInt32}} (alias for PooledArrays.PooledArray{String15, UInt32,
```

Packages

A package is the way that code is distributed as a bundle of add-on functionality to a language. Julia packages contain source code files, tests, documentation, and metadata (e.g., about dependence on other packages).

Using packages

We can make variables/functions from packages available with either `using` or `import`. `using` brings all the variables into the current namespace, while `import` makes them available via `module.variable`. So `import` is cleaner and safer in terms of name conflicts.

I'm having some trouble getting this to display correctly in the rendered document, so the code chunks below have not been run.

```
A = rand(3, 3);
eigvals(A)
```

```
LoadError: UndefVarError: `eigvals` not defined
UndefVarError: `eigvals` not defined

Stacktrace:
 [1] top-level scope
   @ In[4]:2
```

```
import LinearAlgebra
eigvals(A)
```

```
LoadError: UndefVarError: `eigvals` not defined
UndefVarError: `eigvals` not defined

Stacktrace:
 [1] top-level scope
   @ In[5]:2
```

```
LinearAlgebra.eigvals(A)
```

```
3-element Vector{Float64}:
 0.07754956843747109
 0.6969476675364817
 1.429770582715121
```

If we want to access variables/functions from a file, we can use `include`, which will evaluate the file contents in the global scope. This is used a lot in packages to allow the package code to be split up into multiple files.

```
include("linecount.jl")
linecount("notes4.qmd")
```

```
594
```

Installing and managing packages on your system

The examples below show how things work on the SCF Linux machines, but the situation should be similar on your personal machine.

Depots

A depot is where installed packages, projects/environments, and various other things are stored.

You can see the various depots via

```
Base.DEPOT_PATH
```

```
3-element Vector{String}:
 "/accounts/vis/paciorek/.julia"
 "/system/linux/julia-1.10.4/local/share/julia"
 "/system/linux/julia-1.10.4/share/julia"
```

Package installation locations

Julia will look in various locations for packages that are available to be used. The locations can be seen via

```
Base.load_path()
```

```
3-element Vector{String}:
 "/accounts/vis/paciorek/.julia/environments/v1.10/Project.toml"
 "/system/linux/julia-1.10.4/share/julia/stdlib/v1.10"
 "/usr/local/linux/julia-1.10.4/share/julia/environments/v1.10/Project.toml"
```

Note that this includes the active project, the standard library, and (in some cases) a system project. More on projects in a bit.

Julia comes with some packages pre-installed. These include `LinearAlgebra`, `Distributed`, `Random`, and some others.

You can see them in the `stdlib` directory in the main system depot, e.g.,

```
ls /usr/local/linux/julia-1.10.4/share/julia/stdlib/v1.10/
```

Packages that you (or a system administrator) install will be in the `packages` subdirectory of the depots.

For example, for packages you install:

```
ls ~/.julia/packages
```

In some cases (probably not on your personal machine) there will be system-installed packages:

```
ls /usr/local/linux/julia-1.10.4/share/julia/packages
```

As you'd expect, a given package will have various versions, and multiple versions can be installed at the same time.

```
grep version ~/.julia/packages/BenchmarkTools/*/Project.toml
```

In general, Julia will avoid installing multiple copies of the same package version in different locations on a machine/system.

Precompilation

Julia will often precompiled packages. This is separate from runtime just-in-time (JIT) compilation of code. I have not had a chance to understand what the precompilation is doing.

Projects/environments

A *project* (or *environment*) defines a set of packages and their versions. An important aspect of this is reproducibility.

The `Project.toml` file for a project gives high-level information about a project, including the direct package dependencies.

```
cat ~/.julia/environments/v1.10/Project.toml
```

The `Manifest.toml` file gives the detailed dependency information, including package versions for all direct and indirect package dependencies.

```
cat ~/.julia/environments/v1.10/Manifest.toml
```

These files can be handled via version control to ensure reproducibility.

As noted previously, multiple projects can make use of the same installed package version.

`Pkg`: the Julia package manager

The package manager allows you to activate projects and add packages to projects, among other things.

You can use the package manager either in the `Pkg` REPL by pressing `]`, or by calling the `Pkg` API via `Pkg.`, after `using Pkg`. (If you use `]`, you can use the Backspace key to get back to the regular Julia REPL.)

Here we'll use the API as I think that makes the demo clearer.

Activating a project

Anytime you use Julia, you'll be working in the context of a project. By default this will be your default project, which is located at `~/.julia/environments/<version>`.

```
using Pkg

Base.active_project()
Pkg.status()
```

```
Status `~/.julia/environments/v1.10/Project.toml`
  [6e4b80f9] BenchmarkTools v1.5.0
  [078ea971] CPpkg v1.0.0-DEV `/accounts/vis/paciorek/.julia/dev/CPpkg#main`
  [336ed68f] CSV v0.10.14
  [052768ef] CUDA v5.4.2
  [a93c6f00] DataFrames v1.6.1
  [31a5f54b] Debugger v0.7.9
  [31c24e10] Distributions v0.25.109
```

```
  [e2ba6199] ExprTools v0.1.10
  [6a86dc24] FiniteDiff v2.23.1
  [26cc04aa] FiniteDifferences v0.12.32
  [7073ff75] IJulia v1.24.2
  [682c06a0] JSON v0.21.3
  [30363a11] NetCDF v0.12.0
  [14b8a8f1] PkgTemplates v0.7.50
  [91a5bcdd] Plots v1.40.4
  [c46f51b8] ProfileView v1.7.2
  [4c0109c6] QuartoNotebookRunner v0.12.0 `https://github.com/PumasAI/QuartoNotebookRunner.jl.git#mai
  [6f49c342] RCall v0.14.1
  [276daf66] SpecialFunctions v2.4.0
  [a8a75453] StatProfilerHTML v1.6.0
  [fd094767] Suppressor v0.2.7
  [7feac75d] TestPkg v1.0.0-DEV `/accounts/vis/paciorek/.julia/dev/TestPkg#main`
Info Packages marked with  have new versions available and may be upgradable.
```

We can activate (and create if needed) a project (results not shown):

```
Pkg.activate("MyNewProject")
Pkg.status()
```

Adding packages to a project

Before you can invoke `using` or `import` to access a package for the first time in the context of a project, you'll need to "add" the package to the project (but see caveats in the next section).

```
using Pkg
Pkg.add("BenchmarkTools")
# Or to add a specific version:
# Pkg.add(name = "BenchmarkTools", version = "1.5.0")
using BenchmarkTools
```

Julia will prompt you to add a package if it's not part of a project. And if the version of the package is not installed anywhere on your system, it will download and install it (to `~/.julia/packages`). Information about what packages/versions are available and where they are on the internet is contained in a *registry*.

Isolated projects

I don't fully understand the reasoning, but if you activate a project, you still have access to packages from your default project and (if it exists) the system default project. So this seems to make the project not fully isolated. According to the Julia docs, this is because those default projects are *shared environments*.

In the following, note that I never added `JSON` to the active project, but it's accessible from my default project.

```
Base.load_path()
```

```
3-element Vector{String}:
 "/accounts/vis/paciorek/.julia/environments/v1.10/Project.toml"
 "/system/linux/julia-1.10.4/share/julia/stdlib/v1.10"
 "/usr/local/linux/julia-1.10.4/share/julia/environments/v1.10/Project.toml"
```

```
using JSON
```

So good practice when trying to isolate a project is probably to use `Pkg.add` to add any packages you use to your project, rather than relying on those packages being available from elsewhere on the load path. Alternatively you may want to avoid adding packages to the default project entirely.

> Exercise
>
> Create a new project. Add some of the packages that we've used so far to the project (e.g, `Distributions` and `Plots`). Switch back and forth between your new project and the default project. Figure out where the packages are installed on your filesystem.

Creating packages

A package is the way of distributing additional Julia functionality. It generally contains modules (containing code), tests, and documentation. You can also register your package to the Julia General Registry, so that it is easily available via `Pkg.add`.

Creating your package

The Julia documentation provides a nice overview of creating a package. One standard way is to use the `PackageTemplates` package.

```
using PkgTemplates

t = Template(;
            user="paciorek",
            authors=["Christopher Paciorek"],
            plugins=[
                License(name="MIT"),
                Git(),
                GitHubActions(),
            ],
)

t("CPpkg")
```

Running `t(CPpkg)` produces this output, indicating what is happening on the filesystem:

```
julia> t("CPpkg")
[ Info: Running prehooks
```

```
[ Info: Running hooks
  Activating project at `~/.julia/dev/CPpkg`
  No Changes to `~/.julia/dev/CPpkg/Project.toml`
  No Changes to `~/.julia/dev/CPpkg/Manifest.toml`
Precompiling project...
  1 dependency successfully precompiled in 1 seconds
        Info We haven't cleaned this depot up for a bit, running Pkg.gc()...
      Active manifest files: 3 found
      Active artifact files: 139 found
      Active scratchspaces: 5 found
     Deleted 5 package installations (1.827 MiB)
  Activating project at `~/.julia/environments/v1.10`
[ Info: Running posthooks
[ Info: New package is at /accounts/vis/paciorek/.julia/dev/CPpkg
"/accounts/vis/paciorek/.julia/dev/CPpkg"
```

Parts of the package

Code in `src`.

We'll look at the basic structure of code in `src` as outlined in the documentation.

Then we'll add some functionality to `src/CPpkg.jl` or to another file that we include via `include("functions.jl")`.

Git repository

Running the template created a Git repository for the package. Nice.

```
cd ~/.julia/dev/CPpkg
git status
ls -l .git
```

Note that because a Git repository was created, we need to commit any changes to `src` to have them reflected in the package when we use it in Julia (in a moment).

```
git add src/CPpkg.jl
git commit -m'Add initial test function.'
```

Testing

Running the template created a skeleton of a test directory. Also nice.

Even nicer, it created the GitHub Actions workflow (`.github/workflows/CI.yml`) so that your tests would run via continuous integration on GitHub whenever you push changes to the `main` branch of the repository.

Accessing a local package

We can try out the (local) package like this:

```
Pkg.add(path="/accounts/vis/paciorek/.julia/dev/CPpkg")
using CPpkg
test()
```

It looks like we add a package directly from GitHub like this:

```
Pkg.add(url="https://github.com/JuliaLang/Example.jl", rev="master")
```

Looking at an example package

Let's look at the `BenchmarkTools` package as an example package whose structure looks fairly straightforward. The file `src/BenchmarkTools.jl` is the entry point that defines the package and pulls in code from the other source files.

Documentation

It appears that `?BenchmarkTools` shows the contents of `README.md`. There is also information in `docs`, but I haven't figured out the details.

We can see docstrings in the src files though some are fairly brief. `execution.jl` shows the extensive docstring for `@benchmark` though it uses different syntax than seen earlier in these notes, using `@doc raw"""` and `julia-repl`.

Exported objects

If we look in `BenchmarkTools.jl` (in `~/.julia/packages/BenchmarkTools/<version_hash>/src`), we see that `loadparams!` from `parameters.jl` is exported, but `estimate_overhead` is not. The exports define the user interface or "API" of the package.

```
using BenchmarkTools
loadparams!
```

```
loadparams! (generic function with 3 methods)
```

```
estimate_overhead
```

```
LoadError: UndefVarError: `estimate_overhead` not defined
UndefVarError: `estimate_overhead` not defined
```

We can access `estimate_overhead` in the BenchmarkTools module:

```
import BenchmarkTools
BenchmarkTools.estimate_overhead
```

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

Additional modules

One can organize code within a package into multiple modules nested within the main module for the package.

For example, in the LinearAlgebra package (part of Julia's standard library), `src/blas.jl` contains the `BLAS` module. To make it available, `LinearAlgebra.jl` uses `include("blas.jl")` and `export BLAS`. Here's how we can access the BLAS (sub)module:

```
import LinearAlgebra
LinearAlgebra.BLAS.get_num_threads()
```

```
4
```

or

```
using LinearAlgebra
BLAS.get_num_threads()
```

```
4
```

> Exercise
>
> Explore the structure of a package of interest to you in `~/.julia/packages`. (You'll need to add the package to invoke installation if it's not already part of one of your projects.)

Errors and messaging

Tracebacks

We can see the function frames if we force an error to occur.

Here we have a function that uses recursion, but the same thing happens when the nested function calls are not recursive.

```
function factorial(x)
  if x == 0
    sqrt(-1)
    return 1
  else
    return x*factorial(x-1)
  end
end

factorial(3)
```

```
LoadError: DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called with a
DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called with a

Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64)
   @ Base.Math ./math.jl:33
```

```
 [2] sqrt
   @ ./math.jl:686 [inlined]
 [3] sqrt(x::Int64)
   @ Base.Math ./math.jl:1578
 [4] factorial(x::Int64)
   @ Main ./In[18]:3
 [5] factorial(x::Int64) (repeats 3 times)
   @ Main ./In[18]:6
 [6] top-level scope
   @ In[18]:10
```

Finding the source code

If I wanted to dig into the source code to better understand an error or try to understand how it works, here's an example of finding the functions indicated in the traceback:

```
paciorek@gandalf:~> type julia
julia is /usr/local/linux/julia-1.10.4/bin/julia
paciorek@gandalf:~> cd /usr/local/linux/julia-1.10.4
paciorek@gandalf:/usr/local/linux/julia-1.10.4> find . -name math.jl
./share/julia/base/math.jl
./share/julia/test/math.jl
```

Try-catch

One standard way to make your code more robust is to anticipate where errors may occur and give informative messaging and/or proceed despite the error.

```
try
    data = open("bad_file.txt")
catch exc
    println("Something went wrong: $exc")
end
```

```
Something went wrong: SystemError("opening file \"bad_file.txt\"", 2, nothing)
```

Warnings and logging

You can print out useful information with `@info`, `@warn`, and `@error`, going from information messages to more serious issues. The latter doesn't actually throw a formal exception (error) and stop execution, it just indicates a less extreme error from which the code can still proceed.

Users can then control the level of logging information that they see. In this example we set the logging level for warnings and more serious messages, so the Info message is not shown. (This code is not run here because the rendering process is causing problems.)

```
function test()
    println("Hello from test.")
    @info "Some info"
```

```
    @warn "A warning"
    @error "An error"
end

using Logging

# Show warning messages and above (this excludes info messages)
global_logger(ConsoleLogger(stderr, Logging.Warn))

test()
```

These are examples of *macros*, which start with `@`.

Debugging statements

```
x = (3,5)
@show x

@debug "The sum of some values $(sum(rand(100)))"
```

```
x = (3, 5)
```

The debug statement is only run if debugging is enabled (e.g., via `JULIA_DEBUG=all` when invoking Julia or in the Julia interactive session).

```
ENV["JULIA_DEBUG"] = "all"
@debug "The sum of some values $(sum(rand(100)))"
```

```
  Debug: The sum of some values 46.34429078636051
  @ Main In[21]:2
```

We'll see the Julia debugger (and possibly Julia debugging in VS Code) later.

Assertions

One can use assertions to check values for robustness or as sanity checks while developing code.

```
function mysum(x, y)
  @assert(isa(x, Number) && isa(y, Number), "non-numeric inputs")
  return x + y
end
```

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

Of course that's not a good example as we can more robustly and elegantly deal with type checking by declaring types for the function arguments!

For errors that may arise because of user inputs, one generally wants to use exceptions.

Exceptions

Julia has a variety of kinds of runtime errors (aka *exceptions*).

Here's a `DomainError`.

```
sqrt(-1)
```

```
LoadError: DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called with a
DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called with a

Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64)
   @ Base.Math ./math.jl:33
 [2] sqrt
   @ ./math.jl:686 [inlined]
 [3] sqrt(x::Int64)
   @ Base.Math ./math.jl:1578
 [4] top-level scope
   @ In[23]:1
```

And here's how we can trap an error and throw an exception (again a `DomainError`) ourselves.

```
function mysqrt(x)
   if x < 0
      throw(DomainError(x, "sqrt: input must be non-negative."))
   else
      return sqrt(x)
   end
end

mysqrt(-3)
```

```
LoadError: DomainError with -3:
sqrt: input must be non-negative.
DomainError with -3:
sqrt: input must be non-negative.

Stacktrace:
 [1] mysqrt(x::Int64)
   @ Main ./In[24]:4
 [2] top-level scope
   @ In[24]:10
```

Documentation

Docstrings come before the function in triple quotes. Note that I figured out the syntax below (e.g., use of `jldoctest`) by looking at Julia source code.

````
"""
    times3(x)

Multiplies the input by three.

The return type is the same as the input type.

# Examples
```jldoctest
julia> x = 7.5;
julia> times3(x)
22.5

julia> x = [7.5, 3];
julia> times3.(x)
2-element Vector{Float64}:
 22.5
  9.0
```
"""
function times3(x)
  return 3*x
end
````

If you do that, you can then do `?times3` to see the docstring. Nice.

> Tips for docstrings
>
> 1. As many of you have probably discoverd using a Chatbot is a good way to get a first draft of a doc string.
> 2. Following the format/style of docstrings for base Julia functions is a good idea.
> 3. Provide examples! (You know that when you look at docstrings, that's often what you want.)

Testing

We'll cover testing separately.

You can run the tests for an existing package like this:

```
Pkg.test("CSV")
```

Macros (`@`)

Macros modify existing code or generate new code. They're a convenient shortcut to do a variety of things.

We've already seen assertions related to messaging.

Here's a basic example of timing some code:

```
@time 3 + 7;
```

```
  0.000001 seconds
```

The `@.` broadcasting macro 'distributes' the `.` broadcasting to all operations. These two lines of code are equivalent.

```
@. x^2 + y^2  1
x.^2 .+ y.^2 .  1
```

Macros execute when the Julia code is parsed, **before** the code is actually run.