Notes on the Julia Programming Language

Lutz Hendricks

UNC Chapel Hill

August 11, 2020

Abstract

This document summarizes my experience with the Julia language. Its main purpose is to document tips and tricks that are not covered in the official documentation.

1 Introduction

Much of what is collected here is really just that: collected from other sources, which I try to cite where appropriate. Some of the material is a synthesis of material taken from various Discourse threads. In these cases, I may not always recall where I learned something. Apologies to Discourse contributors who I should perhaps have cited, but whose contributions I failed to recall. I will be happy to add citations if omissions are pointed out to me.

2 My Setup (1.5)[1]

My current setup is Julia 1.5 run from the terminal and Visual Studio Code as editor (augmented with BBEdit to overcome VsCode's shortcomings in multi-file search and replace).

Running the `Julia` apps seems identical to running `Julia` from the terminal.

[1] Each section is labeled with the Julia version for which it was last updated.

My startup file loads the packages `OhMyREPL` and `Revise`. `Revise` comes after packages from the standard libraries, so it does not track changes to those.

It appears that the default editor is determined by the system wide file association. No need to set the `JULIA_EDITOR` environment variable.

One drawback: Links in the terminal REPL are not clickable (update: they are when using VsCode).

2.1 Updating to a new version (1.5)

After starting the new version, basic packages need to be added so the startup code can be run (e.g., `OhMyREPL`). In my case these are:

```
]add OhMyREPL Revise
```

The bash profile needs to be updated to point to the new version. Alternatively, create a symlink for the new version with

```
rm /usr/local/bin/julia
ln -s /Applications/Julia-1.5.app/Contents/Resources/julia/bin/julia /u
```

The Jill bash script automates this process.

3 Arrays (1.1)

3.1 Indexing

Extracting specific elements with indices given by vectors:

```
A = rand(4,3,5);
A[CartesianIndex.([1,2], [2,2]), 1] -> A[1,2,1] and A[2,2,1]
```

Similar to using sub2ind:

```
idxV = sub2ind(size(A), [1,2],[2,2],[1,1])
A[idxV]
```

To extract a "row" of a multidimensional matrix without hard-coding the dimensions, generate a `view` using `selectdim`.

To drop dimensions, use `dropdims`.

`LabelledArrays.jl`: allows to assign names to row or column indices or even to slices. This becomes like a Table, but with the performance of a (static) array.

3.2 Broadcasting (1.4)

Treating objects as scalars:

- Generically `f.(x, [1,2,3])` fails because broadcasting tries to iterate over elements of `x`.
- Some objects are treated by default as scalars. Then it works.
- For user defined types, `f.(Ref(x), [1,2,3])` does the trick because a `Ref` is treated as a scalar in broadcasting.
- Or create a broacastable version of `x` with `Base.broadcastable(x :: XType) = Ref(x)` (based on this suggestion).

4 Data Handling (1.1)

`DataFrames` has the rough equivalent of Matlab `table`s.

`JuliaDB` resembles the data handling capabilities of traditional stats packages (like STATA) where processing happens outside of RAM.

4.1 Dataframes (1.1)

Tutorials are collected here.

Stats packages often provide data in "long" format (i.e., each id occupies many rows to cover many variables). unstack and `stack` can go back and forth between "long" and "short" formats.

4.2 STATA files (1.2)

Can be read using StatFiles.jl.

4.3 Data Dependencies

Github issues

- to download from a github repo: link to the "raw" file ("raw/githubusercontent.com/...").
- for private repos, the raw URL ends in a "token"; the presumably handles authorization for access. Or set up an ssh key for each computer and register it with github (instructions at github.com).

`DataDeps.jl`

- each data dependency is a single file (it seems)
- but can download and unpack a tarball

Storing data inside a package:

- directories inside a package can be accessed, even if the code has not been explicitly copied to a computer.
- `@__DIR__` points to the package directory in `.julia/packages`. This does contain copies of all package subdirectories.
- Therefore, one can simply wire the path to the data as `joinpath(@__DIR__, ■..■, data)`. When the package is added somewhere, this will work.

5 Dates and Times (1.2)

Keeping track of elapsed time (e.g., since starting a computation):

- store the start time as a `DateTime` : `startTime = Dates.now()`
- `round(Dates.now() - startTime, Dates.Minute)` gives the elapsed time in minutes.

6 Debugging (1.4)

Julia offers several debugger options that work in very different ways. The main trade-off is compile time versus run time. Debuggers that run interpreted code, such as `Debugger.jl`, compile reasonably fast but run very slowly (about 10 times slower than compiled code). Debuggers that compile debugging features into the code run at near native speed but are either slow to compile (`MagneticReadHead)` or offer limited features (`Infiltrator`).

Common options are:

1. `Debugger.jl`: It interprets all code and can therefore offer a complete feature set. But it is very slow for larger projects. Options are available that make `Debugger` faster, but less powerful. The VS Code plugin gives an IDE experience.

2. `MagneticReadHead.jl`: It compiles debugging features into all code and therefore runs at near native speed. But compile times are often extremely long.

3. `Infiltrator.jl`: It compiles all code, adding only user specified breakpoints (`@infiltrate`). Compile times and run times are good, but the user can only inspect the local state when a break point is reached. It is not possible to move around the call stack.

4. `Exfiltrator.jl`: Simply exports all local variables at the point of call to `Main`. The idea is that the entire local environment can be inspected at really no runtime or compilation cost. Simple and effective, but one cannot manipulate objects in the context of the calling module.

There are other options that I don't know much about, such as `Juno`'s debugger.

6.1 Debugger (1.4)

`Debugger` runs code through `JuliaInterpreter`. By default, everything is interpreted, including code from `Base`. There are two ways of increasing speed:

1. An experimental compiled mode runs code that is stepped over (I am not sure exactly what that means), ignoring breakpoints in that code.
2. One can exclude `Base` (or any other module) code from being interpreted using

```
push!(JuliaInterpreter.compiled_modules, Base);
```

   But note that this steps over any compiled functions, even if they call interpreted functions. Hence, a breakpoint in code that calls `MyModule -> Base -> MyModule` will be missed.

After `using Debugger` invoke `@enter foo(x)` to start a debugging session.
Breakpoints:

- The `@bp` macro does not work in my code (throws unparsable errors or is ignored) but it works in simple examples.
- `break_on(:error)` – does it work in compiled mode?
- `JuliaInterpreter` syntax can also be used to manipulate breakpoints.

Links:

- A useful tutorial
- Discourse thread on debugging

6.1.1 VS Code Interface (1.4)

Gives a graphical interface for setting breakpoints and inspecting variables. A description is here.
Each time code is run, a new debugging window is opened. There is a significant delay until the code starts running.

6.2 Infiltrator (1.4)

A debugger that always runs in compiled mode (hence faster). Breakpoints are set in the code using the `@infiltrate` macro (similar to Matlab's `keyboard` but more limited in capabilities).

Because the code is compiled, one cannot move around in the call stack or change local variables. One can only inspect the local state and then resume that run.

6.3 MixedModeDebugger

Experimental package that tries to speed up Debugger by interpreting only code around breakpoints.

Once a breakpoint is reached, the `Debugger.jl` interface becomes available.

Breakpoints:

- After `using JuliaInterpreter`, its API for manipulating breakpoints becomes available. Breakpoints can be set with the `@breakpoint` macro.
- The `@bp` macro and the `breakpoint()` function do not work.

Example session

```
using JuliaInterpreter, MixedModeDebugger

function foo(x)
  println("foo");
  x += 2
  println(x)
  x += 3
  return x
end

@breakpoint foo(2) 3
@run_mixedmode foo(3)
[...]
>debug
```

6.4 Rebugger

The MacOs keybinding for `interpret` is `Fn-Esc-i`.

7 Documentation (1.3)

`DocStringExtensions.jl` makes it easier to write docs. In particular, function signatures are automatically created in docstrings.

`Documenter.jl` is the package to write documentation.

Literate.jl is useful for generating examples.

A useful guide to writing documentation (pointed out by Tim Holy on Discourse).

8 External Programs (1.3)

One can execute `bash` commands with `run`.

Question: Trying to run a bash script using `run(`. myscript.sh`)` produces a permission denied error (even though permissions are set so that others can execute. Why?

Commands are constructed like strings with interpolation: `` `mkdir $myPath` ``.

When strings are quoted in commands, they are enclosed in single quotes, if they contain spaces.

If the command contains quotes, first build a string and then interpolate it into the command:

```
fPath = "'abc.txt'";
`ls $fPath`
```

For fun (figuring out the answer to a question posed on Discourse): A macro that makes convenient syntax for pipelines

```
julia> macro try1(args...)
        :( pipeline([eval(arg) for arg in $args]...) )
end
```

```
@ppl (macro with 1 method)
julia> @ppl `abc file.jl` `def`
pipeline(`abc file.jl`, stdout=`def`)
julia> @ppl `abc file.jl` `def` "test1.out"
pipeline(pipeline(`abc file.jl`, stdout=`def`), stdout>Base.FileRedirect
```

9 FileIO

`JSON.jl` returns arrays as vectors of vectors of element type `Any`. This makes it difficult to read objects from a file, unless one wants to write custom converters even for standard objects (such as `Array{Float64}`).

9.1 JDF (1.4)

Saves DataFrames. Likely the best option when saving rectangular data (as opposed to serialized objects).

9.2 JSON3 (1.3)

Only one object can be written per file. `JSON3.write` will happily write a vector, but it cannot read it. One has to first make the Vector into a `Dict`.

One drawback of the JSON3 package: types are converted into package specific types. For example, a `Dict{String,Vector}` becomes the JSON3 equivalent `AbstractDict`. In the process, the `Dict keys` are converted from, say, `String` to some hybrid of `String` and `Symbol`. This throws off code that wants to access the loaded objects.

Matrices are stored as Vectors.

9.3 JLD2 (1.3)

Files sizes can be extremely large. In one case, saving an optimization history (stored by `ValueHistories.jl`) increased the file size from 67kb to 3.6GB, even though the `JSON3` history file was only 2MB.

Appears to have been abandoned in mid 2020.

10 Formatted Output

10.1 Formatting individual numbers (1.3)

The `Formatting` package seems to be the best bet. It uses `Python` like syntax and can format multiple arguments simultaneously (not well documented). Example:

```
fs = FormatExpr("{1:.2f} and {2:.3f}")
format(fs, 1.123, 2)
```

yields `"1.12 and 2.000"`.

This cannot be used to format a vector of numbers in one command. Broadcasting also does not work. The easiest approach for this:

```
println(round.(x, digits = 3))
```

10.2 Formatting tables (1.3)

Latex output can be produced with `LatexTables.jl`.

11 Functions and Methods

11.1 Array inputs (1.3)

It is best to restrict inputs to `AbstractArray{T1}` rather than `Array{T1}`. This way, array transformations, such as `reshape`, and `range`s are accepted. For example:

- `typeof(1:4) <: AbstractVector{T1} where T1 <: Integer`

11.2 Keyword arguments (1.4)

Passing keyword arguments through to another function is easy:

```
function foo(x; kwargs...)
        # This is how the args are accessed inside the function
        println(kwargs[:a]);
        # All passed through and expanded into indvidual args. Note the
        bar(x; kwargs...);
end

bar(x; a=1, b=2) = println((a,b));

# Now we can call
foo(1; b=5) == bar(x; a=1, b=5)
```

Allowing a function to ignore "excessive" keyword arguments is also easy:

```
function bar(x; a=1, kwargs...)
        println((x,a));
        println(kwargs);
end
# Can be called with any arguments as long as `x` is provided:
bar(1, b=3)
```

This is useful for functions that pass similar sets of keyword arguments to several sub-functions.

11.2.1 Default arguments

The `CommonLH.KwArgs` type and associated methods are useful for defining default values for keyword arguments.

An easy way of defining defaults is to `merge` with a named tuple:

```
function foo(; kwargs...)
  defaults = (x = 17, );
  args = merge(defaults, kwargs);
  println(args[:x])
end
```

This even works when no args are passed.

12 Installation (MacOS)

Install the `Julia_ver.app` as usual.

Change `bash_profile` to point to the new version's path.

Open Julia from Finder to override MacOS's refusal to start an unknown app.

Exit Julia

To keep previous packages:

```
cp -r ~/.julia/environments/v1.2 ~/.julia/environments/v1.3
```

That also copied registries.

Then it's not a bad idea to run `]pkg up` to get latest versions of packages that are used in `Main`.

13 IO (1.3)

Writing output simultaneously to `stdout` and a file is best accomplished by defining an `IO` struct that contains a vector of `IO` types. Then define `Base.print` to write to all IO streams sequentially. This is implemented in `CommonLH.MultiIO`.

14 Miscellaneous

14.1 Learning Julia

Introductions and guides:

- From Zero to Julia

Useful collections of tips, tricks, and style suggestions:

- How my Julia coding style has changed. Note in particular:
  - using named tuples and `@unpack` for functions that return multiple arguments

  - pointer to `DocStringExtensions.jl`

- Traits

JuliaHub: full text search of package documentation and code.

14.2 Quasi-random numbers

Sobol.jl draws Sobol sequences.

- the same sequence of numbers is drawn each time. The initial condition is not related to the random seed.

14.3 Regular Expressions

Packages that try to simplify constructing Regex: `ReadableRegex.jl` and `RegularExpressions.jl`.

15 Modules

Name conflicts:

- If module A defines `foo` and wants to call the (exported) `foo` from module B, the name needs to be qualified: `B.foo`.
- This makes it advantageous to avoid generic names, such as `name(x)`.

Threads that discuss how to deal with module interdependencies:

15.1 LOAD_PATH (1.1)

Only modules located somewhere along the `LOAD_PATH` can be loaded with `using`.

But: If a directory contains `Project.toml`, it becomes a project directory and only entries listed in `Project.toml` can be loaded (even if the directory is on the `LOAD_PATH`).

As a general rule, though: If one has to fiddle with the `LOAD_PATH`, something is probably not right. Packages have all their dependencies in `Project.toml`. Anything that gets run from the `REPL` is `include`d. That only leaves potential startup code that sits in a `module` as a candidate for being on the `LOAD_PATH`.

15.2 Sub-Modules (1.1)

Functions from sub-modules can be exported by the main module. Example:

```
module scratch
export foo
module inner1
    export foo
    function foo()
        println("foo")
    end
end
using .inner1
end
```

15.3 Extending a function in another module (1.1)

The problem:

- Module `B` defines type `Tb` and function `foo(x :: Tb)`.
- Module `A` contains a generic function `bar(x)` that calls `foo()`. It should use the `foo()` that matches the type of `x`. That is, when called as `foo(x :: Tb)`, we want to call `B.foo`.

Solution:

- Module `A`:
  - Define the stub: `function foo end`
  - Call `foo(x)` from within `bar`.

- Module `B`:
  - Define `function foo(x ::  Tb)`
  - `import A.foo`
- Now `A.bar(x)` knows about `B.foo()` and calls it when the type matches the signature.

See Duck typing when 'quack' is not in 'Base'.

16 Operators

16.1 Logical (1.1)

`&&` is the logical AND operator, but in broadcasting use `.&` (even though `&` is a bitwise AND).

17 Optimization (Mathematical)

JuMP is a popular interface, but it requires (as of v.0.2) analytical derivatives for all objective functions.
Collections:

- NLopt

17.1 NLopt (1.2)

Objective function requires gradient as input, even if it is not used. If gradient is not provided, NLopt returns `FORCED_STOP` without error message.
When objective function errors, return value is `STOPVAL_REACHED` and `fVal=0.0`.

- One way of diagnosing such errors: print the guess to `stdout` at the start of each iteration. Then the objective function can be run again from the REPL with the guesses that cause the crash.
- An alternative (suggested by Kristoffer Carlsson): wrap the entire objective function in `try/catch`. Report the error in the `catch` and then `rethrow` it.

17.2 Noisy objectives

Useful discourse threads: here

SPSA:

- according to the author: specifically made for simulation type problems
- basic idea seems to approximate derivatives, but instead of perturbing each parameter one-by-one (expensive), all are perturbed in the same step.
- extremely easy to implement
- can vary the distribution of step sizes (main algorithm uses step sizes 1 or 2 times a $c(k)$).

COBYLA

- implemented in NLopt COBYLA
- uses a linear approximation of the function

Subplex

- implemented in NLopt Sbplx
- similar to Nelder-Mead, but claims to be more robust

Bayesian optimization

17.3 Global algorithms

QuadDIRECT

- combines ideas of DIRECT with local search
- points from local search are used to form boxes for the global search

NODAL

- global optimization algorithms that can run in parallel

- possibly abandoned

Controlled Random Search

- implemented as NLopt CRS
- starts from a random population of points
- user can control the size of the initial population, but there are no warm starts.
- then evolves these using heuristic rules.

MLSL

- implemented as NLopt MLSL
- basic idea: multistart a local solver, avoiding resolving points that are close to each other

BlackBoxOptim

- implements SPSA
- currently little documentation (2020-May)
- There is an example of distributed parallel optimization. Not clear whether multi-threaded works as well.
- Not clear whether / how optimization history can be saved.

17.4 Surrogate Optimization

Basic idea:

- Sample a small number of points. Evaluate the objective.
- Fit a surrogate function to those points.
- Optimize this function, which is cheap to evaluate, using an algorithm that

  - explores the global parameter space
  - downweights points that are far from points where the true objective has been evaluated

- Add the optimum of the surrogate to the list of evaluated points (using the true function value).
- Update the surrogate model based on the new point.
- Repeat.

Packages:

- Surrogates.jl: a variety of algorithms and sampling methods.
- SurrogateModelOptim.jl

Drawback: There is no good way of running the surrogate optimization in parallel (unlike Matlab).

18 Packages

18.1 Environments (1.3)

An environment is anything with a `Project.toml`. When you start Julia, you enter the version's environment (e.g. 1.3). When you add a package, you effectively edit `Project.toml`.

You can **add** additional environments using `Pkg.activate()` or `pkg> activate .` and then `Pkg.add` to initialize a `Project.toml` in that directory. Now the `Project.toml` of **both** environments are used to resolve dependencies.

The environment determines how code is loaded.

- When you type `using M` Julia looks for module `M` in all directories that are listed in `LOAD_PATH`.

- Julia also looks in the directory of the currently activated package (which is **not** added to the `LOAD_PATH`). Exactly what `Pkg.activate()` does internally is not clear. Once you activate another package, previously activated packages are no longer considered during code loading. But: Whatever was `using`ed previously remains loaded. And since only one version of a given package can be loaded at any point in time, once `M` is loaded its version is fixed, no matter what the `Project.toml` of the current environment says.
- Note: Julia does **not** look in the current directory (unlike Matlab). In fact, the current directory really does nothing at all, except it is the base directory for `REPL` commands such as `cd()` or `include()`.

When examining a particular directory in `LOAD_PATH`, what happens depends on whether the directory contains `Manifest.toml` (or `Project.toml`; two go together).

- If it does not, Julia looks for `M.jl` in this directory.
- Otherwise, Julia **only** looks in `Manifest.toml`. The **only** part is key. Julia does not look in the directory itself.

18.1.1 Stacked environments

When you `activate` an environment, you do **not** deactivate previous environments. Instead, you now operate in a sort of union of all the environments that you activated during a session. This matters when both environments list the same packages in the Manifests.

Example: Start in environment `1.1` and `Pkg.add(D)`. `Pkg.activate(P)` and `Pkg.add(D)` with a different version of `D` (or using the local path for `D`). Which version of `D` is used after `using D`? The answer turns out to be that the most recent environment wins (that would usually be the currently activated project). More precisely, as the documentation explains, code loading merges the manifest entries of the different environments, giving priority to the first entries over later ones (see also here). The active environment is read before the "base" (e.g. `v1.3`) environment, so it wins.

I encountered a case where I could not covince Julia to update an unregistered package, even using `Pkg.rm` followed by `Pkg.add`. The reason was that `1.1` referenced the same package, pointing to a fixed `github commit`.

Only one version of a given package can be loaded at the same time.

If two packages require different versions of the same package, code loading will still work, but running the code may fail because one package is using the "wrong" version of a dependency.

This can lead to interesting problems. Example: I updated a package's dependencies, resulting in a crash. The diagnosis was that `Parsers.jl` was out of date, but the `Manifest` showed the correct version. It turned out that this version was never loaded because a package in my base environment already loaded an older version of `Parsers.jl`. Remember: What is shown in `Manifest.toml` is not necessarily what gets loaded when environments are stacked.

This is one reason why the base environment should be lean and frequently updated.

18.2 Creating a package

18.2.1 PkgTemplates.jl (1.2)

See the Documentation.

18.2.2 PkgSkeleton.jl (1.2)

The easiest way is `PkgSkeleton.jl`. You need to set your `github` info (`user.name` etc) using

```
git config --global user.name YourName
```

This must be done inside a `git` directory. Then `generate` generates the directory structure and the required files (`Project.toml` etc). Example:

```
PkgSkeleton.generate("dir1/MyPackage")
```

Details:

- I first create the repo on github and clone it to the local dir.
- Then I use, from the parent dir:

  ```
  PkgSkeleton.generate("MyPackage", skip_existing_dir = false)
  ```

- This way everything is linked to github from the start.

18.3 Package workflow (1.1)

Your packages will generally be unregistered. Your workflow needs to account for the fact that `Pkg` does not track versions for unregistered packages.

Here are the steps:

1. Initialize a `package` in a folder `pDir`; call the package `P`. This generates a directory structure with `src, test`, etc. If you plan on using this package as a dependency, it is best to place it in a sub-folder of `JULIA_PKG_DEVDIR` (`~/.julia/dev` by default). The reason is that `Pkg.develop` wants to download your code there.

2. While the code is being worked on: `Pkg.activate(ps)`. This makes sure that changes are written to the package's environment (`Project.toml`).

3. To add registered dependencies, simply use `Pkg.add(pkgName)`. No problem.

4. To add unregistered dependencies `D` that may change as you work on your project, use `Pkg.develop` instead.

   (a) Write code that makes a `PackageSpec` for `D`. This simplifies managing the package. Call this `ps`. `ps` should point to `D`'s local directory, not to a `github` url. Otherwise, you end up tracking what is on github rather than your local edits.

   (b) `Pkg.develop(ps)` simply changes the entry for `D` in `Project.toml` from pointing at the `github repo` to pointing at the local dir. Key point: This is only operative while the environment `P` is active.

   (c) `Pkg.develop` is an alternative to `Pkg.add`, which edits `Project.toml` to point at `github`.

5. To freeze the state of the code:

   (a) `push P` and `D` to `github`.

   (b) in the environment for `P`: `Pkg.add(ps)` where `ps` should now point at the `github` url for `D`.

   (c) Even if you continue to push updates for unregistered dependencies to `github`, your package should track the fixed versions (identified by the `sha` key that defines the `commit`). Just don't run `Pkg.update`.

18.4 Unregistered packages as dependencies (1.1)

Important point: Unregistered packages need to be added as dependencies "by hand." `Pkg` cannot track when other packages depend on them. This is a known issue 810. That means:

- Suppose you are working in `P` with dependency `D` that depends on `E`.
- `Pkg.add(D)` does not add `E` to `P`'s `Project.toml`.
- You need to explicitly `Pkg.add(E)`.

Tracking changes in unregistered packages can be done in several ways:

- The solution suggested on discourse suggests to always `develop` packages and to have relative paths in `Manifest.toml`. That would be relative paths of the form `../MyPackage`. User directory expansion, as in `~/abc` does not work.
- `Pkg.add(url = ■https://github.com/myUser/MyPkg■)` dowloads the lastest master and recompiles the code. One option is therefore: run the code on the remote on a new environment. Add each unregistered dependency and then the main package. This is cumbersome, but can be done in a script. The key is to manually add all unregistered dependencies through that script. `Pkg` cannot do so automatically.
- Create your own package registry (not as hard as it sounds). Register all your packages. Then a simple `Pkg.add` for the code that is actually to be run will automatically download all dependencies (which are now registered).

Note:

- `Pkg.update` does nothing for unregistered dependencies.
- Deleting the corresponding subdirectory in `~/.julia/compiled` sometimes triggers a recompile, but not always.
- `revise(MyPkg)` does not trigger a recompile.

For small functions that are themselves stand-alone, it seems best to simply copy them into the project. This is the old trade-off between duplication and dependencies.

18.5 Multiple Modules in one Package (1.2)

The cleanest approach is sub-modules. I.e.,

```
module Foo

include("That.jl")
include("Bar.jl")

using .Bar, .That

<code>

end

# In Bar.jl
module Bar
        using ..That
        <code>
end
```

One can still `import Foo.Bar` to only use the sub-module (especially for testing). In the test function, non-exported functions can be called as `Bar.f()`.

The alternative is to split the package into multiple packages. To run this on a HPC, the `dev`'d packages need to be included as relative paths in Project.toml. And the code for all of the packages needs to be copied to the HPC.

18.6 Testing a package (1.2)

Activate the package by issuing `activate .` in the package's directory (not in `src`). Then type `test`.

Note that the package needs the following in `Project.toml`:

```
[extras] Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"
[targets] test = ["Test"]
```

These are not automatically added. You need to hand-edit `Project.toml`. Or simply add `Test` as a dependency directly.

Placing test code inside a module:

- This can be useful when the test code defines `structs` that one would like to be able to modify without having to restart `Julia` all the time. Note that objects defined in tests are no longer visible once `Pkg` is exited.
- Place the module definition into `test`. Add `push(LOAD_PATH, @__DIR__)`. This has to be done in each module. Not elegant. This apparently no longer works in V1.2. It appears that tests cannot be in modules any more.

18.7 Creating a package registry (1.2)

Any registry that lives in `~/.julia/registries` is automatically used by `Pkg`.

In principle, it is easy to create your own registry (see discourse for a guide). The key to making it practical is `LocalRegistry.jl`.

18.7.1 Using LocalRegistry.jl (1.3)

This is the successor to Gunnar Farneback's `Registrator.jl`.

Once a package has been registered with `Registrator.jl`, registering a new version simply requires `register(MyPackage; commit = true, push = true)`. The keyword arguments push a git commit of the local registry to github.

This directly edits the copy of the registry in `.julia/registries`.

18.7.2 Using `Registrator.jl` (1.2)

This is based on the workflow figured out by a discourse user. What I am writing up here copies their code almost one-for-one.

Creating the registry:

```
Pkg.add(Pkg.PackageSpec(url="https://github.com/GunnarFarneback/Registra
using Registrator
# Switch to the parent directory of the registry repo
```

```
cd(joinpath(homedir(), "Documents", "julia"))
regName = "registryLH"
regUrl = "https://github.com/hendri54/$regName"
regPath = joinpath(homedir(), "Documents", "julia", "registryLH")
Registrator.create_registry(regName,  regUrl)
```

Create the `registryLH` repo on `github.com` (not sure why this is necessary). Push the registry to github using

```
git -C registryLH push -u -f origin master
```

Check that `Registry.toml` appears on github in the repo. Add the registry (cloning it to `.julia/registries`)

```
Pkg.Registry.add(Pkg.RegistrySpec(url = regUrl))
```

We now have an empty registry. Check that it can be used:

```
Pkg.update()
```

This should now show `registryLH` being updated.

Note: For Julia 1.4 you should replace the "GunnarFarneback" clone by LocalRegistry.jl. Things appear to be very much the same then. Thanks to Juergen Fuhrmann for pointing this out.

Adding packages to the registry. I am using `TestPkgLH` for testing.

```
# Somehow get to the point where `using TestPkgLH` can be issued
pkgName = "TestPkgLH"
pkgRepo = "https://github.com/hendri54/TestPkgLH"
# Somehow Julia knows where this is located (how?)
Pkg.activate(pkgName)
using TestPkgLH

# Register TestPkgLH
Registrator.register(TestPkgLH, regPath)
run(`git -C registryLH push origin master`)
# Check on github that T/TestPkgLH appears in the repo
```

```
# Now we wish to `add` TestPkgLH to TestPkg2LH
pkgName2 = "TestPkg2LH"
pkgRepo2 = "https://github.com/hendri54/TestPkg2LH"
Pkg.activate(pkgName2)

# Without the `update` the `add` fails
Pkg.Registry.update()
# Make sure that the latest version of TestPkgLH has been pushed to gith
Pkg.add(pkgName)
using TestPkg2LH
Registrator.register(TestPkg2LH, regPath)
run(`git -C registryLH push origin master`)
Pkg.Registry.update()
```

It is currently not possible to run this from inside a module. For some reason, the wrong `Registrator` is called (or `Registrator` has no methods). So one has to do this "by hand" from the `REPL` :

```
julia> activate_pkg("UtilityFunctionsLH")
Activating environment at `~/Documents/julia/UtilityFunctionsLH/Project
julia> regPath = joinpath(homedir(), "Documents", "julia", "registryLH")
"/Users/lutz/Documents/julia/registryLH"
julia> isdir(regPath)
true
julia> using UtilityFunctionsLH
julia> register(UtilityFunctionsLH, regPath)
# These last two steps are done by `PackageToolsLH.update_registry()`
# Push to github
julia> Pkg.Registry.update()
```

I packaged this into a function which is used as follows:

```
using TestPkgLH
include("shared/register_package.jl")
register_package(TestPkgLH)
```

Updating an existing package works in the same way. Increment the version number in `Project.toml` and register the package again.

18.8 Relocatable Packages (1.4)

18.8.1 Data Dependencies

18.9 Miscellaneous

Find unused dependencies
Find out if a package is installed (present in current `Manifest`):

- `d = Pkg.installed()` returns a `Dict` with package names as keys.
- `haskey(d, MyPackage)` returns true if package is installed.

Adding a private repo works the same way as a public repo. But one has to first add an ssh key and register it with github.
Multiple packages depend on the same package.

- The question: which version of the dependency gets used?
- Answer (due to Gunnar Farneback): "The general idea is that packages state which versions of their dependencies they are compatible with and it's the job of the resolver to find a set of versions that works for all packages in the dependency chain. Another key idea is that only one version of each package can be loaded at a time, which is precisely the one that the resolver has chosen. If there are incompatible version requirements in the dependency chain the resolver will fail and you can't load your package at all."

Updating packages breaks `Revise`'s tracking. For updates to take effect, Julia needs to be restarted.
When a package fails to `instantiate`, register a new version and run `Pkg.update`.

19 Parallel Computing

Useful overviews: Bruel 2019

19.1 Threads

Use shared memory. Simply place `Threads.@threads` in front of a code section (typically a loop).

Runs on a single processor (with multiple cores).

Julia needs to be started with a command line argument that indicates the number of cores to use. Or issue `export JULIA_NUM_THREADS=8` in shell (not persistent across sessions unless written into `bash_profile`).

Keeping track of progress:

19.2 Printing in multi-threaded code

Standard `print` statements are not displayed (or written to a log file) until the entire computation finishes.

This discourse thread suggests that the solution is to use `Core.println` (as opposed to `Base.println` which is not thread-safe). Note that `Core.println` does not accept `stdout` as argument.

Could also try to `flush(stdout)` periodically.

To print output in the correct sequence: one solution is this discourse thread: use a `SpinLock` together with (thread-safe) `Core.println`.

One solution for keeping track of long computations: Write output to a file instead of (or in addition to) stdout. File output is written in the correct order. Periodically `flush` to ensure that output is written and the file can be viewed. This basically replaces the log files the the cluster managers would usually generate (redirecting `stdout`).

19.3 Distributed computation

Issue `using Distributed, SharedArrays`.

In front of the parallel loop, place `@sync @distributed`. The `@sync` macro ensures that the code waits for completion of the loop before it continues.

20 Performance

20.1 Optional asserts (1.5)

ToggleableAsserts.jl offers a version of assert that can be switched on and off globally.

One can get no-ops conditional asserts using

```
dbg() = false;
dbg() && @assert <stuff>
```

Does the same work with `const Dbg`?

20.2 Profiling (1.3)

The output generated by the built-in profiler is hard to read.

`ProfileView` now does compile, taking a surprisingly long time. Personally, I find the presentation of `StatProfilerHTML` more convenient, though.

`StatProfilerHTML` is a good alternative (1.1).

- It provides a flame graph with clickable links that show which lines in a function take up most time.
- Need to locate `index.html` and open it by hand in the browser after running `statprofilehtml()`.

`PProf.jl`:

- requires `Graphviz`. On MacOS, install using `brew install graphviz`. But it has TONS of dependencies and did not install on my system. Then PProf cannot be used.

`TimerOutputs.jl`

- can be used to time selected lines of code
- produces a nicely formatted table that is much easier to digest than profiler output.

20.3 Type stability

One can automate checking for type stability using the `code_warntype()` function. Example:

- For function `foo(x)`, call `code_warntype(stdout, foo, (Int,1))`.
- This can be written to a file by changing the `IO` argument.
- It generates output even if no issues are found.
- The amount of output generated is overwhelming. Signs of trouble are `Union` types, especially return types (at `Body:`).

Cthulhu.jl is a tool for debugging type instability.

20.4 Tricks

1. Avoid allocations in calls to functions like `any()`. For example, `any(x .< 5)` allocates while `any(z -> z < 5, x)` does not (and is much faster).

21 Plotting

Visually, `PlotlyJS` produces the most appealing plots (for me). But it does not install on my system (1.3).

When a plotting related library is not found (as in "error compiling display"), try `]build Plots`.

21.1 Legends

The `label` is set when each series is plotted. If `labels` are set when the plot is created (before the series are plotted), the entries are ignored.

21.2 Saving data with plots:

VegaLite does this natively.

with `Plots.jl` one can use `hdf5plot_write` to write an entire plot, including the data, to an hdf5 file.

- This means that each plot has to be generated twice; once with whatever backend is used to generate PDF files; and then again with `hdf5`. In particular, one cannot first plot with another backend and then save the resulting plot object to `hdf5`.
- The approach is then to first save the plot to `hdf5`, then load it and save it with another backend.

Note: In my current (v.1.3) installation, `hdf5plot_write` generates a bunch or warnings followed by a crash due to world age problems.

22 Regressions

RegressionTables.jl produces formatted regression tables.

22.1 GLM (1.2)

GLM.jl is the package to run regressions.

To save just the regression results (without the data, which could be a lot of memory), use `coeftable(mdl)`. This produces a `StatsBase.CoefTable`.

Alternative, use `RegressionTable` from `EconometricsLH`.

Categorical regressors return names such as `Symbol(■school: 3■)`.

A useful introduction is in cookbooks.

23 Remote Clusters

23.1 Getting started with a test script

How to get your code to run on a typical Linux cluster?

- Get started by writing a simple test script (`Test3.jl`) so we can test running from the command line.
- Add the Julia binary to the `PATH` using (on Macos, editing `~/.bash_profile`):

  ```
  PATH="/Applications/Julia-1.1.app/Contents/Resources/julia/bin:$PATH"
  ```

- Then make sure you can run the test script locally with `julia ■/full/path/to/Test3.jl■`

Now copy `Test3.jl` to a directory on the cluster and repeat the same.

- You may need to add the Julia binary to the path.
  - On Longleaf (editing `~/.bash_profile`): `export PATH="/nas/longleaf/apps/julia/1.3.0/bin:$PATH"`
  - The more robust approach is `module add julia/1.3.0`.
- Then run `julia "/full/path/to/Test3.jl"`

Now run the test script via batch file:

```
sbatch -p general -N 1 -J "test_job" -t 3-00 --mem 16384 -n 1 --
mail-type=end --mail-user=lhendri@email.unc.edu -o "test1.out" -
-wrap="julia /full/path/to/Test3.jl"
```

23.2 Generate an ssh key

This allows log on without password. Instructions on the web.

Now you can use the terminal to log in with `ssh user@longleaf.unc.edu`.

23.3 Rsync File Transfer

A reliable command line transfer option is `rsync`. The command would be something like

```
rsync -atuzv "/someDirectory/sourceDir/" "username@longleaf.unc.edu:someDirectoryS
```

Notes:

- The source dir should end in "/"; the target dir should not.

- Exluding `.git` speeds up the transfer.
- `--delete` ensures that no old files remain on the server.

To transfer an individual file: `run(‘scp $filename hostname:/path/to/newfile.txt’)`

23.4 Git File Transfer

1. Change into the package directory (which is already a `git repo`).
2. Add a remote destination (once):
   `git remote add longleaf ssh://lhendri@longleaf.unc.edu/nas/longleaf/home/lhen`
3. Initialize the remote directory with a bare repo: `git init --bare`. Bare means that the actual files are not copied there. It needs to be bare so `push` does not produce errors later.
4. Verify the remote: `git remote show longleaf`

When files have changed:

1. Change into the package directory
2. `git commit -am ■commit message■`
3. `git push longleaf master`

Note that this does not upload any files! So this only works for packages, not for code that should be run outside of packages.

Note: Perhaps a bare `git repo` is not needed after all.

23.5 Running code on the cluster

Steps:

1. Copy your code and all of its dependencies to the cluster (see Section 23.3). This is not needed when all dependencies are registered.
2. Write a Julia script that contains the startup code for the project and then runs the actual computation (call this `batch.jl`).
3. Write a batch file that submits `julia batch.jl` as a job to the cluster's job scheduler. For UNC's `longleaf` cluster, this would be slurm. So you need to write `job.sl` that will be submitted using `sbatch job.sl`.

23.5.1 The Julia script

Submitting a job is (almost) equivalent to `julia batch.jl` from the terminal.

- Note: `cd()` does not work in these command files. To include a file, provide a full path.

If you only use registered packages, life is easy. Your code would simply say:

```
using Pkg
# This needs to be run only once
Pkg.add(MyPackage)
# If you want the latest version each time
Pkg.update()
using MyPackage
MyPackage.run()
```

If the code for `MyPackage` has been copied to the remote, then

```
julia --project="/path/to/MyPackage" --startup-file=no batch.jl
```

activates `MyPackage` and runs `batch.jl`. The `--project` option is equivalent to `Pkg.activate`.

- Julia looks for `batch.jl` in the directory that was active when Julia was invoked (in this case: when `sbatch` was invoked).
- Disabling the `startup-file` prevents surprises where the `startup-file` changes the directory before looking for `batch.jl`.
- `~` is not expanded when relative paths are used.

If `MyPackage` contains is unregistered or contains **unregistered dependencies**, things get more difficult. Now `batch.jl` must:

1. Activate the package's environment.
2. `develop` all unregistered dependencies. This replaces the invalid paths to directories on the local machine (e.g. `/Users/lutz/julia/...`) with the corresponding paths on the cluster (e.g. `/nas/longleaf/...`). Note: I verified that one cannot replace `homedir()` with `~` in `Manifest.toml`.

3. `using MyPackage`
4. `MyPackage.run()`

Developing `MyPackage` in a blank folder does not work (for reasons I do not understand). It results in errors indicating that dependencies of `MyPackage` could not be found.

This approach requires you to keep track of all unregistered dependencies and where they are located on the remote machine. My way of doing this is contained in `PackageTools.jl` in the `shared` repo (this is not a package b/c its very purpose is to facilitate loading of unregistered packages).

For an example implementation of the entire process, see `batch_commands.jl` in `TestPkg2LH`.

- This uses `PackageToolsLH` to handle directories on different computers and file transfer.
- `write_command_file()` writes the julia file that is to be executed remotely (`command_file.jl`).
- `write_sbatch` writes the sbatch file that will be submitted to `slurm`.
- `project_upload()` uses `rsync` to copy the code of the project, its dependencies, and some general purpose code that is required at startup (mainly `PackageToolsLH` itself) to the remote machine.

23.5.2 The sbatch file

How this works can be looked up online. The only trick is that the Julia command requires a full path (or a relative path, but that's a little risky) on the remote machine.

`FilesLH` keeps track of where things are on each machine. It is used to build the full paths.

23.5.3 Instantiating Packages

If all dependencies are registered, simply activate an environment and `>pkg add https://github.com/user/MyPackage.git` followed by `using MyPackage`.

When packages are run, all dependencies must be installed. This would usually be done with `instantiate`. But this fails when the package is `develop`ed rather than `add`ed. Therefore: if a package fails to build or test (for example, after its first upload, or after new dependencies are installed that the remote machine does not have installed):

1. An indicator that a dependency is missing is the error message: `ERROR: MethodError: Cannot ‘convert‘ an object of type Nothing to an object of type Base.SHA1`
2. Switch to a test environment where one can mess up the `Project.toml`.
3. `Pkg.add(ps)` where `ps` is the `PackageSpec` for the package that does not build. It must point at the `github` url.
4. This is not always enough. In that case, `activate` the package that does not build. Use `>pkg st -m` to show the packages that are not loaded and simply `add` them until the package builds and tests.

Now the package can be built or developed everywhere.

Sometimes old versions of `Project.toml` lie around somewhere (where?) in the Julia installation. They may contain dependencies that don't exist anymore. Then the package does not build. The only solution that seems to work: `Pkg.add` the package from somewhere with a `PackageSpec` that points at `github`.

- For this purpose, it is useful to have an environment lying around that is just for adding packages that need to be downloaded.

24 Types (1.3)

Parametric types without the type parameter are NOT `DataTypes`; they are `UnionAll`.

- Example: `struct Foo{T} end; isa(Foo, DataType) == false;`

I find it easiest to write model specific code NOT using parametric types. Instead, I define `type aliases` for the types used in custom types (e.g., `Double=Float64`). Then I hardwire the use of `Double` everywhere. This removes two problems:

1. Possible type instability as the compiler tries to figure out the types of the custom type fields.
2. It becomes possible to call constructors with, say, integers of all kinds without raising method errors.

24.1 Constructors (1.4)

Constructing objects with many fields:

- Define an inner constructor that leaves the object (partially) uninitialized. It is legal to have new(x) even if the object contains additional fields.

`Parameters.jl` is useful for objects with default values.

- Constructor must then provide all arguments that do not have defaults.
- Note that `@with_kw` automatically defines `show()`. Use `@with_kw_noshow` to avoid this.
- `Base.@kwdef` now does much of the same.

24.2 Inheritance (1.1)

There is no inheritance in Julia. Abstract types have no fields and concrete types have no subtypes.

There are various discussions about how to implement types that share common fields.

For simple cases, it is probably best to just repeat the fields in all types. This can be automated using `@forward` in `Lazy.jl`.

One good piece of advice: ensure that methods are generally defined on the abstract type, so that all concrete types have the same interface (kind of the point of having an abstract type).

24.3 Loading and saving (1.4)

Loading and saving user defined types is a **major problem** in Julia. There is, to my knowledge, no working solution right now.

JLD

- currently maintained (2020-July)
- unable to save functions (Issue 57)

JLD2

- `using FileIO` and extension `.jld2` automatically saves in `jld2` format. This can save used defined types.
- fails to load files without any apparent pattern (UnsupportedFeatureException)
- saving user objects converted to `Dicts` does not help (InvalidDataException).
- All modules needed to construct the loaded types need to be known in the loading module and in `Main`. See Issue 134. It is not possible to use `Core.eval(Main, :(using Module))` for unclear reasons.

`BSON.jl`

- Note that `BSON` modifies Dicts when they are saved.
- Fails to load files without any apparent pattern.

24.3.1 Saving ModelObjects

This refers to saving `ModelObjects` defined in `ModelParams`.

The most feasible solution is:

1. Construct the object with default parameters.
2. Load a `Vector{ParamVector}`.

3. Apply the values of the calibrated parameters (according to the object's own `ParamVector`) and the default values of the not-calibrated parameters.

How to save/load a `Vector{ParamVector}` is not so obvious. But one only has to save the values. The other info is determined at model construction.

24.4 Named Tuples

Creating a named tuple where names are based on variables:

```
x = :abc;
nt = (; x => 100)
# returns (abc = 100,)
```

25 Unit Testing (1.2)

Goals:

1. Ensure that tests are self-contained, so that each can be run independently.

My current approach:

1. Place each `testset` inside a function.
2. Call these functions from within other `testset`s.
3. One can now `include` each file and run the tests independently.
4. The function provide some isolation (similar to using modules).

Module approach:

1. Place each group of tests into a module, so the tests are independent of each other and can be run independently. `SafeTestsets.jl` has a similar idea, but I find it cleaner to explicitly write out the modules. Though modules have the benefit that they can `include` setup code that is used repeatedly in different tests.

2. `runtests.jl` simply contains a list of `include` statements; one for each test module. Those are wrapped in a `@testset` for nice display and to ensure that errors don't stop the tests.

3. Each test module also contains a `@testset`.

4. When `runtests` is run, it displays a single success summary. But when there are errors, they are nicely broken down by `testset`.

5. To run tests selectively, simply `include` the file that contains the `@testset` at the `REPL`.

Errors in the code to be tested (but not caught by `@test`) cause the entire test run to crash. Preventing this requires all tests to be enclosed in a `@testset`. A sequence of `@testset` does not do the trick. An error in one prevents all others from being run. Nested `@testset`s produce nested error reports (nice).

`@test` statements can be placed inside functions. To preserve result reporting, the function should contain a @testset and return its result.

Test dependencies now need to be added to the `Project.toml` file in `./test`.

25.1 Travis CI (1.2)

Travis can automatically test all branches uploaded to github.

Need to customize `travis.yml` to only build for the current Julia version.

Building with unregistered dependencies is tricky. Probably ok if the dependencies are added (so they point to a github url), but not if they are `develop`ed.

26 Workflow (1.2)

`Revise` is key. It is now possible to simply use `using` on any `module` once. `Revise` then automatically keeps track of changes. Using `includet` creates problems for me.

But keep in mind that `Revise` cannot handle:

1. changes in file structure (you factor out some code into a new file that is `include`d in the main file);

2. changes in `structs`.

Those still require restarting the `REPL`.
