Jesse Perla, Thomas J. Sargent and John Stachurski

July 23, 2020

1 Contents

- Preliminary Setup 2
- The REPL 3
- Atom 4
- Package Environments 5

Co-authored with Arnav Sood

While Jupyter notebooks are a great way to get started with the language, eventually you will want to use more powerful tools.

We’ll discuss a few of them here, such as

- Text editors like Atom, which come with rich Julia support for debugging, documentation, git integration, plotting and inspecting data, and code execution.
- The Julia REPL, which has specialized modes for package management, shell commands, and help.

Note that we assume you’ve already completed the getting started and interacting with Julia lectures.

2 Preliminary Setup

Follow the instructions for setting up Julia on your local computer.

3 The REPL

Previously, we discussed basic use of the Julia REPL (“Read-Evaluate-Print Loop”).

Here, we’ll consider some more advanced features.
3.1 Shell Mode

Hitting ; brings you into shell mode, which lets you run bash commands (PowerShell on Windows)

In [1]: ; pwd

/home/ubuntu/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia

You can also use Julia variables from shell mode

In [2]: x = 2

Out[2]: 2

In [3]: ; echo $x

2

3.2 Package Mode

Hitting ] brings you into package mode.

- ] add Expectations will add a package (here, Expectations.jl).
- Likewise, ] rm Expectations will remove that package.
- ] st will show you a snapshot of what you have installed.
- ] up will (intelligently) upgrade versions of your packages.
- ] precompile will precompile everything possible.
- ] build will execute build scripts for all packages.
- Running ] preview before a command (i.e., ] preview up) will display the changes without executing.

You can get a full list of package mode commands by running

In [4]: ] ?

Welcome to the Pkg REPL-mode. To return to the julia> prompt, either press backspace when the input line is empty or press Ctrl+C.

Synopsis

pkg> cmd [opts] [args]

Multiple commands can be given on the same line by interleaving a ; between the commands.

Commands

activate: set the primary environment the package manager manipulates
add: add packages to project
build: run the build script for packages
develop: clone the full package repo locally for development
free: undoes a pin, develop, or stops tracking a repo
gc: garbage collect packages not used for a significant time
generate: generate files for a new project
help: show this message
instantiate: downloads all the dependencies for the project
pin: pins the version of packages
precompile: precompile all the project dependencies
redo: redo the latest change to the active project
remove: remove packages from project or manifest
resolve: resolves to update the manifest from changes in dependencies of developed packages
status: summarize contents of and changes to environment
test: run tests for packages
undo: undo the latest change to the active project
update: update packages in manifest
registry add: add package registries
registry remove: remove package registries
registry status: information about installed registries
registry update: update package registries

On some operating systems (such as OSX) REPL pasting may not work for package mode, and you will need to access it in the standard way (i.e., hit ] first and then run your commands).

3.3 Help Mode

Hitting ? will bring you into help mode.
The key use case is to find docstrings for functions and macros, e.g.

? print

Note that objects must be loaded for Julia to return their documentation, e.g.

? @test
will fail, but

using Test

? @test

will succeed.

4 Atom

As discussed previously, eventually you will want to use a fully fledged text editor. The most feature-rich one for Julia development is Atom, with the Juno package.

There are several reasons to use a text editor like Atom, including

- Git integration (more on this in the next lecture).
- Painless inspection of variables and data.
- Easily run code blocks, and drop in custom snippets of code.
- Integration with Julia documentation and plots.

4.1 Installation and Configuration

4.1.1 Installing Atom

1. Download and Install Atom from the Atom website.

2. (Optional, but recommended): Change default Atom settings

   - Use Ctrl-, to get the Settings pane
   - Choose the Packages tab
   - Type line-ending-selector into the Filter and then click “Settings” for that package
     - Change the default line ending to LF (only necessary on Windows)
   - Choose the Editor tab
     - Turn on Soft Wrap
     - Set the Tab Length default to 4

4.1.2 Installing Juno

1. Use Ctrl-, to get the Settings pane.

2. Go to the Install tab.

3. Type uber-juno into the search box and then click Install on the package that appears.

4. Wait while Juno installs dependencies.

5. When it asks you whether or not to use the standard layout, click yes.

At that point, you should see a built-in REPL at the bottom of the screen and be able to start using Julia and Atom.
4.1.3 Troubleshooting

Sometimes, Juno will fail to find the Julia executable (say, if it’s installed somewhere non-standard, or you have multiple).

To do this 1. Ctrl-\- to get Settings pane, and select the Packages tab. 2. Type in julia-client and choose Settings. 3. Find the Julia Path, and fill it in with the location of the Julia binary.

- To find the binary, you could run Sys.BINDIR in the REPL, then add in an additional /julia to the end of the screen.
- e.g. C:\Users\YOURUSERNAME\AppData\Local\Julia-1.0.1\bin\julia.exe on Windows as /Applications/Julia-1.0.app/Contents/Resources/julia/bin/julia on OSX.

See the setup instructions for Juno if you have further issues.

If you upgrade Atom and it breaks Juno, run the following in a terminal.

apm uninstall ink julia-client
apm install ink julia-client

If you aren’t able to install apm in your PATH, you can do the above by running the following in PowerShell:

cd $ENV:LOCALAPPDATA/atom/bin

Then navigating to a folder like C:\Users\USERNAME\AppData\Local\atom\bin (which will contain the apm tool), and running:

./apm uninstall ink julia-client
./apm install ink julia-client

4.1.4 Upgrading Julia

To get a new release working with Jupyter, run (in the new version’s REPL)

] add IJulia
] build IJulia

This will install (and build) the IJulia kernel.

To get it working with Atom, open the command palette and type “Julia Client: Settings.”

Then, in the box labelled “Julia Path,” enter the path to your Julia executable.

You can find the folder by running Sys.BINDIR in a new REPL, and then add the /julia at the end to give the exact path.

For example:
4.2 Standard Layout

If you follow the instructions, you should see something like this when you open a new file.

If you don’t, simply go to the command palette and type “Julia standard layout”

The bottom pane is a standard REPL, which supports the different modes above. The “workspace” pane is a snapshot of currently-defined objects. For example, if we define an object in the REPL

\[
\text{In [5]: } x = 2
\]

\[
\text{Out[5]: } 2
\]

Our workspace should read

The \texttt{ans} variable simply captures the result of the last computation. The \texttt{Documentation} pane simply lets us query Julia documentation
The **Plots** pane captures Julia plots output (the code is as follows)

```julia
using Plots
gr(fmt = :png);
data = rand(10, 10)
h = heatmap(data)
```

**Note:** The plots feature is not perfectly reliable across all plotting backends, see [the Basic Usage page](#).

### 4.3 Other Features

- **Shift + Enter** will evaluate a highlighted selection or line (as above).
• The run symbol in the left sidebar (or Ctrl+Shift+Enter) will run the whole file.

See basic usage for an exploration of features, and the FAQ for more advanced steps.

5 Package Environments

Julia’s package manager lets you set up Python-style “virtualenvs,” or subsets of packages that draw from an underlying pool of assets on the machine.

This way, you can work with (and specify) the dependencies (i.e., required packages) for one project without worrying about impacts on other projects.

• An environment is a set of packages specified by a Project.toml (and optionally, a Manifest.toml).

• A registry is a git repository corresponding to a list of (typically) registered packages, from which Julia can pull (for more on git repositories, see version control).

• A depot is a directory, like ~/.julia, which contains assets (compile caches, registries, package source directories, etc.).

Essentially, an environment is a dependency tree for a project, or a “frame of mind” for Julia’s package manager.

• We can see the default (v1.1) environment as such

```
In [6]:  ] st

Status  '~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/Project.toml`
[2169fc97] AlgebraicMultigrid v0.2.2
[28f2ccd6] ApproxFun v0.11.13
[7d9fca2a] Arpack v0.4.0
[aee01518] BandedMatrices v0.15.7
[6e4b80f9] BenchmarkTools v0.5.0
[a134a8b2] BlackBoxOptim v0.5.0
[ff5ab731] BlockBandedMatrices v0.8.4
[324d7699] CategoricalArrays v0.8.0
[34da2185] Compat v2.2.0
[a93c6f90] DataFrames v0.21.0
[1313f7d8] DataFramesMeta v0.5.1
[39dd38d3] Dierckx v0.4.1
[9ffde737] DiffEqOperators v4.10.0
[31c24e10] Distributions v0.23.2
[2fe49d83] Expectations v1.1.1
[a1e7a1ef] Expokit v0.2.0
[d4d017d3] ExponentialUtilities v1.6.0
[442a2c76] FastGaussQuadrature v0.4.2
[1a297f60] FillArrays v0.8.9
[9d5c8c9] FixedEffectModels v0.10.7
[c8885935] FixedEffects v0.7.3
[587475ba] Flux v0.10.4
[f6369f11] ForwardDiff v0.10.10
[38e38edf] GLM v1.3.9
[28b8d3ca] GR v0.49.1
[40713840] IncompleteLU v0.1.1
[43edad99] InstantiateFromURL v0.5.0
[a989a8b] Interpolations v0.12.9
[b6b21f68] Ipopt v0.6.1
[42fd00bc] IterativeSolvers v0.8.4
```
• We can also create and activate a new environment

**In [7]:**  generate ExampleEnvironment

    Generating project ExampleEnvironment:
    ExampleEnvironment/Project.toml
    ExampleEnvironment/src/ExampleEnvironment.jl

• And go to it

**In [8]:**  ; cd ExampleEnvironment
To activate the directory, simply

```
In [9]: ] activate .
```

Activating environment at `~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/ExampleEnvironment/Project.toml`

where “~” stands in for the “present working directory”.

Let’s make some changes to this

```
In [10]: ] add Expectations Parameters
```

Updating registry at `~/.julia/registries/General`

Resolving package versions...

```
    Updating `~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/ExampleEnvironment/Project.toml`
      [2fe49d83] + Expectations v1.4.0
      [d96e819e] + Parameters v0.12.1
    Updating `~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/ExampleEnvironment/Manifest.toml`
      [7d9fc1a2a] + Arpack v0.4.0
      [68821587] + Arpack_jll v3.5.0+3
      [9e28174c] + BinDeps v1.0.1
      [b99e7846] + BinaryProvider v0.5.10
      [34da2185] + Compat v3.13.0
      [e66e0078] + CompilerSupportLibraries_jll v0.3.3+0
      [9a962f9c] + DataAPI v1.3.0
      [864ed93b] + DataStructures v0.17.19
      [31c24e10] + Distributions v0.22.6
      [2fe49d83] + Expectations v1.4.0
      [442a2c76] + FastGaussQuadrature v0.4.2
      [1a297f60] + FillArrays v0.8.13
      [e1d29d7a] + Missings v0.4.3
      [4536629a] + OpenBLAS_jll v0.3.9+5
      [bac5587e] + OrderedCollections v1.3.0
      [90014a1f] + PDMats v0.9.12
      [d96e819e] + Parameters v0.12.1
      [1fd47b50] + QuadGK v2.4.0
      [70009fc4] + Rmath v0.6.1
      [f50d1b31] + Rmath_jll v0.2.2+1
      [a2af1166] + SortingAlgorithms v0.3.1
```
Note the lack of commas

- To see the changes, simply open the ExampleEnvironment directory in an editor like Atom.

The Project TOML should look something like this

```toml
name = "ExampleEnvironment"
uuid = "14d3e79e-e2e5-11e8-28b9-19823016c34c"
authors = ["QuantEcon User <quanteconuser@gmail.com>"]
version = "0.1.0"

[deps]
Expectations = "2fe49d83-0758-5602-8f54-1f90ad0d522b"
Parameters = "d96e819e-fc66-5662-9728-84c9c7592b0a"
```

We can also

```
In [11]: ] precompile
```

```
Precompiling project...
  Info: Precompiling Expectations [2fe49d83-0758-5602-8f54-1f90ad0d522b]
  @ Base loading.jl:1260
  Warning: Package Expectations does not have LinearAlgebra in its dependencies:
      - If you have Expectations checked out for development and have
        added LinearAlgebra as a dependency but haven't updated your primary
```
environment's manifest file, try `Pkg.resolve()`.
- Otherwise you may need to report an issue with Expectations

Loading LinearAlgebra into Expectations from project dependency, future warnings for
Expectations are suppressed.

```
Info: Precompiling ExampleEnvironment [845f4396-974c-4930-a296-0333d89d5adf]
```

Note The TOML files are independent of the actual assets (which live in

You can think of the TOML as specifying demands for resources, which are supplied by the
~/.julia user depot.

- To return to the default Julia environment, simply

```
In [12]: ] activate

Activating environment at `~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/Project.toml`
```

without any arguments.

- Lastly, let’s clean up

```
In [13]: ; cd ..

/home/ubuntu/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia
```

```
In [14]: ; rm -rf ExampleEnvironment
```

### 5.1 InstantiateFromURL

With this knowledge, we can explain the operation of the setup block

```
In [15]: using InstantiateFromURL
# optionally add arguments to force installation: instantiate = true, precompile = true
  github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

What this `github_project` function does is activate (and if necessary, download, instantiate and precompile) a particular Julia environment.