Julia Quick Syntax Reference

A Pocket Guide for Data Science Programming

—

Antonello Lobianco

Apress®

Julia Quick Syntax Reference

A Pocket Guide for Data Science Programming

Antonello Lobianco

Apress®

Julia Quick Syntax Reference: A Pocket Guide for Data Science Programming

Antonello Lobianco
Nancy, France

ISBN-13 (pbk): 978-1-4842-5189-8   ISBN-13 (electronic): 978-1-4842-5190-4
https://doi.org/10.1007/978-1-4842-5190-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484251898. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Antonello Lobianco, PhD is a research engineer employed by a French Grande école (Polytechnic University). He works on biophysical and economic modeling of the forest sector and is responsible for the Lab Models portfolio. He uses C++, Perl, PHP, Python, and Julia. He teaches environmental and forest economics at the undergraduate and graduate levels and modeling at the PhD level. He has been following the development of Julia as it fits his modeling needs, and he is the author of several Julia packages (search for sylvaticus on GitHub for more information).

About the Technical Reviewer

Germán González-Morris is a polyglot software architect/engineer with 20+ years in the field. He has knowledge of Java (EE), Spring, Haskell, C, Python, and JavaScript, among others. He works with web distributed applications. Germán loves math puzzles (including reading Knuth) and swimming. He has tech reviewed several books, including an application container book (Weblogic), as well as titles covering various programming languages (Haskell, TypeScript, WebAssembly, Math for Coders, and RegExp, for example). You can find more information on his blog (`https://devwebcl.blogspot.com/`) or Twitter account (`@devwebcl`).

Acknowledgments

This work has been supported by the French National Research Agency through the Laboratory of Excellence, ARBRE, part of the “Investissements d’Avenir” program (ANR 11 – LABX-0002-01).

I want to thank Germán González-Morris for his valuable help in finding errors in the code and improving the description of the language. I want to also thank Mark Powers, the Apress coordinating editor, for his numerous “check ins” that pushed me to continue and finish the book.

This has been possible thanks to the understanding and support of my family.

Introduction

This Julia quick syntax reference book covers the main syntax elements of the Julia language as well as some of its more important packages.

The first chapter explains the basic skills needed to set up the software you need to run and develop Julia programs, including managing Julia packages.

Chapter 2 presents the many predefined types (integers, strings, arrays, etc.) and the methods to work with them. Memory and copy issues are also presented in this chapter, together with an important discussion about the implementation of the various concepts of *missingness*.

After the basic data types have been introduced, Chapter 3 deals with how to organize them in a sequence of logical statements to compose your program. Control flow, functions, blocks, and scope are all discussed in this chapter.

In Chapter 4, we extend our discussion to custom types—in Julia, both primitive and composite types can be custom-defined—and to their organization in the program, either using inheritance or composition. This chapter will be of particular use to readers accustomed to other object-oriented programs, in order to see how to apply object-oriented concepts in Julia.

Chapter 5 explains how to retrieve the inputs needed by your program from a given source (the terminal, a text/CSV/Excel/JSON file, or a remote resource) and conversely, to export the outputs of your program.

In Chapter 6, we discuss a peculiar feature of Julia, that is, the possibility to manipulate the code itself after it has been parsed, but before it is compiled and run. This paves the way to powerful macro programming. We discuss it and present the concepts of *symbols* and *expressions* in Chapter 6.

Julia is a relatively new language, and while the package ecosystem is developing extremely rapidly (as most packages can be written directly in the Julia language alone), it is highly likely that you will still need libraries for which a direct port to Julia is not yet available. Conversely, your main workflow may be in another, slower, high-level language and you may want to use Julia to implement some performant-critical tasks. Chapter 7 shows how to use C, C++, Python, and R code and their relative libraries in Julia and, conversely, embed Julia code in Python or R programs.

The following chapter (Chapter 8) gives a few recommendations for writing efficient code, with runtime performances comparable to compiled languages. We also deal here with *programmer's efficiency*, discussing profiling and debugging tools and with a short introduction to runtime exceptions.

This completes the discussion of the *core* of the language. Julia, however, has been designed as a thin language where most features are provided by external packages, either shipped with Julia itself (a sort of Julia Standard Library) or provided by third parties.

Therefore, the second part of the book deals with this Julia package ecosystem. Chapter 9 introduces the main packages for working with numerical data: storage with data structure packages like `DataFrames` and `IndexedTables`; munging with `DataFramesMeta`, `Query`, and `Pipe`; and visualization with the `Plot` package.

If Chapter 9 deals with processing numerical data, Chapter 10 deals with mathematical libraries for more theoretical work. `JuMP` is an acclaimed "algebraic modeling language" for numerical optimization (and can be in itself the primary reason to learn about Julia). We present two complete examples with linear and non-linear models. The second model is then rewritten to be analytically resolved with `SymPy`, which is a library for symbolic computation, e.g. the analytical resolution of derivatives, integrals, and equations (and systems of equations). Chapter 10 ends with a presentation of `LsqFit`, a powerful and versatile library to fit data. Finally, Chapter 11 concludes the book with a series of tools that are of more general use, like composing dynamic documents with `Wave`, dealing with ZIP files with `ZipFile`, and exposing a given Julia model on the web with `Interact` and `Mux`. Examples given in the text are intentionally trivial. They are minimal working examples of the syntax for the concepts they explain. If you are looking for recipes directly applicable to your domain, a "cookbook" kind of book may be more convenient.

While each package has been tested specifically with Julia 1.2 and 1.3-rc4, thanks to the Julia developers' commitment to a stable API, they should remain relevant for the entire 1.x series. Concerning third-party packages, we report the exact version we tested our code with. The section entitled "Using the Package Manager" in Chapter 1 explains how to update a package to a given version if subsequent versions of the package break the API.

Is such cases, please report the problem to us using the form at `https://julia-book.com`. We will regularly publish updates and errata on this site, where a discussion forum focused on the book is also available.

PART I

Language Core

CHAPTER 1

Getting Started

1.1 Why Julia?

With so many programming languages available, why create yet another one? Why invest the time to learn Julia? Is it worth it?

One of the main arguments in favor of using Julia is that it contributes to improving a trade-off that has long existed in programming—fast coding versus fast execution.

On the one side, Julia allows the developer to code in a dynamic, high-level language similar to Python, R, or MATLAB, interacting with the code and having powerful expressivity (see Chapter 6, for example).

On the other side, with minimum effort, developers can write programs in Julia that run (almost) as fast as programs written in C or FORTRAN.

Wouldn't it be better, though, to optimize existing languages, with their large number of libraries and established ecosystems, rather than create a new language from scratch?

Well, yes and no. Attempts to improve runtime execution of dynamic languages are numerous. PyPy (`https://pypy.org`), Cython (`https://cython.org`), and Numba (`https://numba.pydata.org`) are three notable examples for the Python programming language. They all clash with one fact: Python (and, in general, all the current dynamic languages) was designed before the recent development of just-in-time (JIT) compilers, and hence it offers features that are not easy to optimize. The optimization tools either fail or require complex workarounds in order to work.

Conversely, Julia has been designed from the ground up to work with JIT compilers, and the language features—and their internal implementations—have been carefully considered in order to provide the programmer with the expected productivity of a modern language, all while respecting the constraints of the compiler. The result is that Julia-compliant code is guaranteed to work with the underlying JIT compiler, producing in the end highly optimized compiled code.

The Shadow Costs of Using a New Language

If it is true that the main "costs" of using a new language relate to learning the language and having to abandon useful libraries and comfortable, feature-rich development editors that you are accustomed to, it is also true that in the Julia case these costs are mitigated by several factors:

- The language has been designed to syntactically resemble mainstream languages (you'll see it in this book!). If you already know a programming language, chances are you will be at ease with the Julia syntax.
- Julia allows you to easily interface your code with all the major programming languages (see Chapter 7, "Interfacing Julia with Other Languages"), hence reusing their huge sets of libraries (when these are not already ported to Julia).

- The development environments that are available—e.g., Juno (`https://junolab.org`), IJulia Jupiter kernel (`https://github.com/JuliaLang/IJulia.jl`), and VSCode Julia plugin (`https://github.com/JuliaEditorSupport/julia-vscode`)—are frankly quite cool, with many common features already implemented. They allow you to be productive in Julia from the first time you code with it.

Apart from the breakout in runtime performances from traditional high-level dynamic languages, the fact that Julia was created from scratch means it uses the best, most modern technologies, without concerns over maintaining compatibility with existing code or internal architectures. Some of the features of Julia that you will likely appreciate include built-in Git-based package manager, full code introspection, multiple dispatches, in-core high-level methods for parallel computing, and Unicode characters in variable names (e.g., Greek letters).

Thanks to its computational advantages, Julia has its natural roots in the domain of scientific, high-performance programming, but it is becoming more and more mature as a general purpose programming language. This is why this book does not focus specifically on the mathematical domain, but instead develops a broad set of simple, elementary examples that are accessible even to beginner programmers.

1.2 Installing Julia

Julia code can be run by installing the Julia binaries for your system available in the download section (`http://julialang.org/downloads/`) of the Julia Project website (`https://julialang.org`).

The binaries ship with a Julia interpreter console (aka, the "REPL"—Read, Eval, Print, Loop), where you can run Julia code in a command-line fashion.

For a better experience, check out an Integrated Development Environment, for example, Juno (`http://junolab.org/`) an IDE based on the Atom (`https://atom.io`) text editor, or IJulia (`https://github.com/JuliaLang/IJulia.jl`), the Julia Jupiter (`http://jupyter.org/`) backend.

Detailed setup instructions can be found on their respective sites, but in a nutshell, the steps are pretty straightforward.

- For Juno:
    - Install the main Julia binaries first.
    - Download, install, and open the Atom text editor (`https://atom.io`).
    - From within Atom, go to the Settings ➤ Install panel.
    - Type `uber-juno` into the search box and press Enter. Click the Install button on the package with the same name.
- For IJulia:
    - Install the main Julia binaries first.
    - Install the Python-based Jupyter Notebook server using the favorite tools of your OS (e.g., the Package Manager in Linux, the Python spip package manager, or the Anaconda distribution).
    - From a Julia console, type `using Pkg; Pkg.update();Pkg.add("IJulia");Pkg.build("IJulia")`.

- The IJulia kernel is now installed. Just start the notebook server and access it using a browser.

You can also choose, at least to start with, not to install Julia at all, and try instead one of the online computing environments that support Julia. For example, JuliaBox (`https://juliabox.com/`), CoCalc (`https://cocalc.com/doc/software-julia.html`), Nextjournal (`https://nextjournal.com`), and Binder (`https://mybinder.org`).

Some tricks for Juno and IJulia

- Juno can:
  - Enable block selection mode with `ALT` + `SHIFT`.
  - Run a selection of code by selecting it and either selecting Run Block or typing `SHIFT` + `Enter` on Windows and Linux or `CMD` + `Enter` on Mac.
  - Comment/uncomment a block of code with `CTRL` + `/` (Windows and Linux) or `CMD` + `/` (Mac).
- IJulia:
  - Check out the many keyboard shortcuts available from Help ➤ Keyboard Shortcuts.
  - Need to run Julia in a computational environment for a team or a class? Use JupyterHub (`https://github.com/jupyterhub/jupyterhub`), the multi-user solution based on Jupyter.

1.3 Running Julia

There are many ways to run Julia code, depending on your needs:

1. Julia can run interactively in a console. Start `julia` to obtain the `REPL` console, and then type the commands there (type `exit()` or use CTRL+D when you are finished).

2. Create a script, i.e. a text file ending in `.jl`, and let Julia parse and run it with `julia myscript.jl [arg1, arg2,..]`.

   Script files can also be run from within the Julia console. Just type `include("myscript.jl")`.

3. In Linux or on MacOS, you can instead add at the top of the script the location of the Julia interpreter on your system, preceded by `#!` and followed by an empty row, e.g. `#!/usr/bin/julia` (You can find the full path of the Julia interpreter by typing `which julia` in a console.). Be sure that the file is executable (e.g., `chmod +x myscript.jl`).

   You can then run the script with `./myscript.jl`.

4. Use an Integrated Development Environment (such as those mentioned), open a Julia script, and use the run command specific to the editor.

You can define a global (for all users of the computer) and local (for a single user) Julia file that will be executed at any startup, where you can for example define functions or variables that should always be available. The location of these two files is as follows:

- Global Julia startup file: `[JULIA_INSTALL_FOLDER]\etc\julia\startup.jl` (where `JULIA_INSTALL_FOLDER` is where Julia is installed)
- Local Julia startup file: `[USER_HOME_FOLDER]\.julia\config\startup.jl` (where `USER_HOME_FOLDER` is the home folder of the local user, e.g. `%HOMEPATH%` in Windows and ~ in Linux)

Remember to use the path with forward slashes (`/`) with Linux. Note that the local `config` folder may not exist. In that case, just create the `config` folder as a `.julia` subfolder and start the new `startup.jl` file there.

Julia keeps all the objects created within the same work session in memory. You may sometimes want to free memory or "clean up" your session by deleting no longer needed objects. If you want to do this, just restart the Julia session (you may want to use the trick mentioned at the end of Chapter 3) or use the `Revise.jl` (`https://github.com/timholy/Revise.jl`) package for finer control.

You can determine which version of Julia you are using with the `versioninfo()` option (within a Julia session).
