Introduction to `julia`

G. Jay Kerns

March 2, 2013

Contents

This document is a *brief* introduction to `julia`. It is a reworked *Brief Introduction to R* (which is an abbreviated Chapter 2 of IPSUR) which I usually distribute to students using R for the first time. One of the reasons for this document is that I wanted to get better acquainted with `julia` and thought it might make it easier for others to get better acquainted with it, too. In what follows, we assume you have at least a passing familiarity with Org-mode and Emacs keybindings.

Please bear in mind that the discussion below is written as if a person has launched `julia` with `M-x julia` and is sitting in front of an ESS `julia` interactive session right now. But that isn't required. In fact, one of the best things about Org-mode is that such a thing in fact **is not** required, and a person can happily breeze through an org file tapping `C-c C-c` as (s)he goes. For the purpose of this introductory document it doesn't matter which approach you use; be warned, however, that some of the lower code blocks depend on values from the upper code blocks and if you try to execute them out of order then you will get an error.

1 What you need to get started

You will need to install `julia`, a recent version of ESS, and an updated version of Org-mode. Instructions for how to accomplish all three tasks are detailed in Org-mode and `julia`: an introduction. Please read that document first to get up to speed before continuing.

Note: several code blocks below have the header argument `:eval no-export` which means that the code block can be evaluated interactively in this session by `C-c C-c` with point in the code block but will *not* be evaluated during export. The reason is that those blocks have settings which conflict with my current setup yet are meant to be useful for others people.

2 Getting started with julia

2.1 Communicating with julia

There are three basic methods (provided by ESS) for communicating with `julia`.

- **An Interactive session (julia>).** This is the most basic way to complete simple, one-line commands. Do `M-x julia RET` during an Emacs session and the Emacs/ESS `julia` mode will open in a buffer. Type whatever command you like; `julia` will evaluate what is typed there and output the results in the buffer. This method is akin to launching `julia` in a terminal.

- **Source files.** For longer programs (called *scripts*) there is too much code to write all at once in an interactive session. Further, sometimes we only wish to modify a small piece of the script and before running it again in `julia`.

  The way to do this is to open a dedicated `julia` script buffer with the sequence `C-x C-f whatever.jl`, where `whatever.jl` is a `julia` script which you've named whatever. Write the code in the buffer, then when satisfied the user evaluates lines or regions according to the following table. Then `julia` will evaluate the respective code and give output in the interactive buffer.

`C-RET`	Send region or current line and step to next line of code.
`M-C-x`	Send region or function or paragraph.
`C-c C-c`	Send region or function or paragraph and step to next line.

- **Script mode.** It is also possible to write a `julia` script (say, `myscript.jl` and evaluate the script from a terminal like this:

  ```
  ~$ julia myscript.jl
  ```

  Depending on what's in the script, `julia` can be instructed to do all sorts of things. See `julia --help` for additional options. (Note: you will need to add `julia` to your `PATH` or otherwise execute `julia` from its location before proceeding with this method.)

2.2 `julia` is one fancy calculator

`julia` can do any arithmetic you can imagine. For example, in an interactive session type `2 + 3` and observe

```
2 + 3
```

```
5
```

The `julia>` means that `julia` is waiting on your next command. Entry numbers will be generated for each row, such as

```
[3:50]
```

```
48-element Int32 Array:
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
```

```
  ⋮
 41
 42
 43
 44
 45
 46
 47
 48
 49
 50
```

Notice that `julia` doesn't show the whole list of numbers, it elides them with vertical ellipses ⋮. Note also the `[3:50]` notation, which generates all integers in sequence from 3 to 50. One can also do things like

```
2 * 3 * 4 * 5  # multiply
sqrt(10)       # square root
pi             # pi
sqrt(-2)

120
3.1622776601683795
3.141592653589793
ERROR: DomainError()
 in sqrt at math.jl:111
```

Notice that a `DomainError()` was produced; we are not allowed to take square roots of negative numbers. Also notice the number sign `#`, which is used for comments. Everything typed on the same line after the `#` will be ignored by julia. There is no `julia` continuation prompt. If you press `RET` before a statement is complete then empty lines keep piling up until you finish the command.

Some other fuctions that will be of use are `abs()` for absolute value, `log()` for the natural logarithm, `exp()` for the exponential function, and `factorial()` for... uh... factorials.

Assignment is useful for storing values to be used later. Notice the semicolon at the end of the first statement. Without the semicolon, `julia` would print the result of the assigment (namely, `5`).

```
y = 5;    # stores the value 5 in y
3 + y
```

```
8
```

There aren't other assignment operators (like `<-` in R). For variable names you can use letters (perhaps followed by) numbers, and/or underscore "_" characters. You cannot use mathematical operators, you cannot use dots, and numbers can't go in front of numbers (those are interpreted by `julia` as coefficients). Here are some valid variable names: `x`, `x1`, `y32`, `z_var`.

If you would like to enter the data 74,31,95,61,76,34,23,54,96 into `julia`, you may create a data array with double brackets (the analogue of the `c()` function in R).

```
fred = [74, 31, 95, 61, 76, 34, 23, 54, 96]
```

```
9-element Int32 Array:
 74
 31
 95
 61
 76
 34
 23
 54
 96
```

The array `fred` has 9 entries. We can access individual components with bracket `[ ]` notation:

```
fred[3]
fred[2:4]
fred[[1, 3, 5, 8]]
```

```
95
3-element Int32 Array:
 31
 95
 61
```

```
4-element Int32 Array:
 74
 95
 76
 54
```

Notice we needed double brackets for the third example. If you would like to empty the array `fred`, you can do it by typing `fred = []`.

Data arrays in `julia` have type. There are all sorts of integer types (`Int8`, `uInt8`, `Int32`, ...), strings (`ASCIIString`), logical (`Bool`), unicode characters (`Char`), then there are floating-point types (`Float16`, `Float32`), even complex numbers like `1 + 2im` and even rational numbers like `3//4`, not to mention `Inf`, `-Inf`, and `NaN` (which stands for *not a number*). If you ever want to know what it is you're dealing with you can find out with the `typeof` function.

```
simpsons = ["Homer", "Marge", "Bart", "Lisa", "Maggie"];
typeof(simpsons)
```

```
Array{ASCIIString,1}
```

Here is an example of a logical vector:

```
x = 5;
x >= 6
```

```
false
```

Notice the >= symbol which stands for "greater than or equal to". Many functions in `julia` are vectorized. Once we have stored a data vector then we can evaluate functions on it.

```
sum(fred)
length(fred)
sum(fred)/length(fred)
mean(fred)   # sample mean, should be same answer
```

```
544
9
60.44444444444444
60.44444444444444
```

Other popular functions for vectors are `min()`, `max()`, `sort()`, and `cumsum()`.

Arithmetic in `julia` is usually done element-wise, and the operands (usually) must be of conformable dimensions.

```
mary = [4, 5, 3, 6, 4, 6, 7, 3, 1];
fred + mary
fred - mean(mary)
```

```
9-element Int32 Array:
 78
 36
 98
 67
 80
 40
 30
 57
 97
9-element Float64 Array:
 69.6667
 26.6667
 90.6667
 56.6667
 71.6667
 29.6667
 18.6667
 49.6667
 91.6667
```

The operations `+` and `-` are performed element-wise. Notice in the last vector that `mean(fred)` was subtracted from each entry in turn. This is also known as data recycling. Other popular vectorizing functions are `sin()`, `cos()`, `exp()`, `log()`, and `sqrt()`.

An operation which is *not* performed elementwise is array multiplication, `*`. If were were to try `fred*mary` then we would get an error:

```
ERROR: no method *(Array{Int32,1},Array{Int32,1})
```

The reason for the error is that `julia` is trying to do matrix multiplication on two `9x1` arrays which, we know from Linear Algebra, is not allowed. Instead, we can accomplish element-wise multiplication with the following:

```
fred.*mary
```

```
9-element Int32 Array:
 296
 155
 285
 366
 304
 204
 161
 162
  96
```

Notice the dot before the multiplication. A similar trick works for elementwise division. By the way, with two `9x1` arrays it is legal to compute the dot product like this:

```
fred'*mary
```

```
1-element Int32 Array:
 2029
```

where notice we have transposed `fred` and done ordinary matrix multiplication with `fred'*mary`.

2.3 Getting Help

When you get in the thick of `julia` you will soon find yourself looking for help. The help resources for `julia` are not (yet) as extensive as those for some other languages that have been around for a while (such as R). `julia` is new and many of the help topics haven't been written yet. Nevertheless, sometimes a person is lucky and you can get help on a function when it's available with the `help()` function.

```
help("factorial")
```

```
Base.factorial(n)

   Factorial of n

Base.factorial(n, k)

   Compute "factorial(n)/factorial(k)"
```

In addition to this, you can type `help()` which gives an extended list of help topics. For instance, I find myself doing `help("Statistics")` a lot.

3 Other tips

It is unnecessary to retype commands repeatedly, since Emacs/ESS remembers what you have entered at the `julia>` prompt. To navigate through previous commands put point at the lowest command line and do either `M-p` or `M-n`.

3.1 Other resources

- Check out the official `julia` manual here.
- The *Standard Library* (a different type of manual) is here.
- There is a vibrant and growing `julia` community whose gateway is here.
- There is a large and growing list of contributed packages here.

4 Plotting with Winston

There's a pretty well fleshed out plotting example in the Graphics section of Org-mode and `julia`: an introduction. Check it out.

5 Fitting (generalized) linear models

Douglas Bates (of Mixed Effects Models in S and S-PLUS fame) has been putting together a `julia` package called GLM which already supports fitting generalized linear models to datasets. This, together with the RDatasets package means there is already a bunch of stuff to keep a person busy. Below is a modified example from the Multiple Regression chapter of IPSUR, translated to `julia` speak.

First, we start `using` the packages we'll need.

```
using RDatasets, DataFrames, Distributions, GLM
```

Next we load the `trees` data frame from the RDatasets package (via the DataFrames package) and fit a linear model to the data.

```
trees = data("datasets", "trees")
treeslm = lm(:(Girth ~ Height + Volume), trees)
```

There is a *ton* of output from both the above commands which we omit here for the sake of brevity. Most of it, though, is similar to to output we might see in an R session. We can extract the model coefficients with the `coef` function:

```
coef(treeslm)

3-element Float64 Array:
 10.8164
 -0.0454835
  0.19518
```

and we can finish by looking at a summary table similar to something like `summary(treeslm)` in R.

```
coeftable(treeslm)

3x4 DataFrame:
          Estimate Std.Error  t value   Pr(>|t|)
[1,]       10.8164    1.9732  5.48165 7.44691e-6
[2,]    -0.0454835 0.0282621 -1.60935   0.118759
[3,]       0.19518 0.0109553  17.8161 8.2233e-17
```