Julia in VS Code #2
Basic Julia commands

Chris Griffiths, Eva Delmas

November 18, 2020

This doc follows on from "Using Julia in VS code #1" and assumes that your still working from your activated directory.

This doc covers the following:

- Arrays and Matrices
- DataFrames and CSVs
- Functions
- Loops
- Plots
- Scoping

There is also a section at the end with some "Quick tips"

First, load the packages that you will need for this tutorial, and set a random seed:

```
using Plots, DataFrames, Distributions, Random, DelimitedFiles, RDatasets, Gadfly
Random.seed!(33)
```

Note: to check the documentation of functions, go to the REPL, type `?` to enter the help mode (you now should see `help?>` instead of `julia>`) and type the function name. You can use tab to autocomplete or double type tab to suggest all functions that start with what you have already type. Try this to find the difference between `print`and `println`.

1 Allocating objects

```
# Allocate an integer:
number = 5
# Allocate a floating point number:
pi_sum = 3.1415
# note: pi can also be called using pi or π (type \pi and tab to transform into the
unicode symbol)
pi_sum2 = pi
pi_sum3 = π
```

```
# you can actually use unicode symbols in Julia, this can be useful for naming
parameters following standards:
λ = 4
# and you can attribute multiple variables at the same time:
αi, βi, γi = 1.3, 2.1, exp(39)
αi
```

1.1 Checking object types

```
t = typeof(number) #you can use typeof to identify the type of an object
# Note - Julia is like R and Python, it can infer the type of object (Integer, Float)
on the left hand side of the equals sign, you don't have to justify it like you do in C.
# However, you can if needed e.g.
pi_sum1 = Float64(3.141592)
# (5) You can then check the object type using:
typeof(pi_sum), typeof(pi_sum2) # note here that by using the preallocated variable pi,
we are actually using an object of type Irrational (a specific type of float)
```

1.2 Strings

```
aps = "Animal and Plant Sciences" # must use "" and not ''
# Note that you can easily insert variables into strings using $ and concatenate
strings using * :
tp, tp2 = typeof(pi_sum), typeof(pi_sum2)
println("While pi_sum is a $tp" * ", pi_sum2 is an $tp2")
# by using the preallocated variable pi, we are actually using an object of type
Irrational (a specific type of float)
# You can print an object (useful when running simulations) using:
print(aps) # prints in the same line or
println(aps) # prints on the next line (useful for printing in loops, etc)
```

1.3 Array and matrices

```
# A one-dimensional array (or vector, alist of ordered data with a shared type) can be
specified using:
ar = [1,2,3,4,5] # square brackets act like c() in R
br = ["Pint", "of", "moonshine", "please"]
```

You can access the different position by using indexing:

```
ar[1]
br[3]
```

You can use some functions to pre-allocate vectors with a certain value Pre-allocating vectors with the right dimension and type helps to speed things up in Julia

```
emptyvec = zeros(10) #this create a vector of length 10 filled with 0.
onesvec = ones(10) #can be done with ones
boolvec = trues(10) #or true/false (with falses(10))
```

You can also initialise an array that can contain any amount values using:

```
I_array = []
J_array = Float16[] #you can also pass the type you want to store
```

You don't have to provide a nrow or size argument like you would you R. It is also worth noting that this I_array object behaves very similar to list() in R and can handle many different forms - for example it can store matrices or text

You can also use built in commands to construct sequences of numbers (same as seq() or range() in R)

```
range_array = range(0, 10, length = 11) # sequence from 0-10 with length 11
range_array2 = [0:1:10] #alternative
typeof(range_array) #note that this produces an object of type StepRange and note a
vector
# you can turn it into a vector by "collecting"
range_collected = collect(range_array)
range_collected2 = [0:1:10;] #Alternatively you can use the ';' as a last argument here
to automatically collect
# Note that one of these method produces an array of Integers, the other of Floats
typeof(range_collected2)
# Both collect() and range() are useful when constructing a loop, as are the length()
and unique() commands. Both length() and unique() operate the same way they do in R.
```

To apply a function over a vector (or matrix), use the "." operator, this is called broadcasting:

```
exp_array = exp10.(range_array) # Here, the . maps the exp10 function to all elements of
range_array
```

You can bind elements to an array (useful when constructing a parameter matrix for BEFW simulations):

```
dr = append!(ar, 6:10) #see also push!
```

1.4 DataFrames and CSVs

1.5 Functions

1.5.1 Simple mathematical functions

```
c = 2
d = 3
sumcd = c + d
diffcd = c - d
productcd = c * d
divcd = c / d
powcd = c^2
```

For a complete list of all mathematical operations see [https://docs.julialang.org/en/v1/manual/mathema operations/index.html][https://docs.julialang.org/en/v1/manual/mathematical-operations/index.html]

1.6 Loops

1.7 Plots

1.8 Scoping

2 Converting

You can convert object from one type to another (when it is possible) using the `convert` function.

```
a = 2
b = convert(Float64, a)
b = Float64(a) #also works
#converting arrays:
convert(Array{Float64,1}, range_collected) # conversion from 1D array of Integers to 1D
array of floats
```

3 Simple operations