Basic Julia commands

by Chris Griffiths, Eva Delmas and Andrew Beckerman, Dec. 2020.

This document follows on from "Getting started" and assumes that you're still working in your active project.

This document 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".

Load packages

You'll need a few packages for this tutorial:

```
using Plots, DataFrames, Distributions, Random, DelimitedFiles, RDatasets, Gadfly, CSV
```

```
MersenneTwister(UInt32[0x00000021], Random.DSFMT.DSFMT_state(Int32[532514924, 10735227
```

```
# Random.seed - important for consistency. Sets the starting number used to generate a sequence of random numbers - ensures that you get the same result if you start with that same seed each time you run the same process.
Random.seed!(33)
```

All of these packages are listed in the tutorial's `Manifest.toml` and `Project.toml` files, if you're having problems, head back to the script at the bottom of "Using Julia in VS Code #1" and rerun. If you're unsure, remember to use the status command `] st` to check what packages are currently active in your project.

Before we move on, one helpful thing to note are the help files and how to access them. As in R, the help files assiocated with a given package, function or command can be accessed using `?` followed by the function name (e.g. type `? pi` in the REPL). Similar to when you entered Julia's package manager (using `]`) you'll notice that the `?` command causes a change in the REPL with `help?>`

replacing `julia>`, this informs you that you've entered the help mode. As an exercise, use the help mode to find the difference between `print` and `println`.

Preamble

Before we start creating arrays and matrices, we'd like to demonstrate how you allocate numbers and strings to objects in Julia and check an object's type. We'd also like to highlight some simple mathematical operations.

Allocating objects

Allocating in Julia is useful as it means that variables can be stored and used elsewhere. You allocate numbers to objects using the following:

```
nu = 5
```

```
# allocate an Integer:
nu = 5
```

```
pi_sum = 3.1415
```

```
# allocate a Floating point number:
pi_sum = 3.1415
```

```
pi_sum2 = π = 3.1415926535897...
```

```
# 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 = 3.1415926535897...
```

```
pi_sum3 = π
```

```
λ = 4
```

```
# you can also use unicode symbols directly in Julia, this can be useful for naming parameters following standards:
λ = 4
```

```
(1.3, 2.1, 8.65934e16)
```

```
# and you can attribute multiple variables at the same time, useful when outputing from a function:
αi, βi, γi = 1.3, 2.1, exp(39)
```

```
1.3
```

```
αi
```

Strings

You can also allocate strings (text) to objects:

```
aps = "Animal and Plant Sciences"
```

```
aps = "Animal and Plant Sciences" # must use "" and not ''
```

```
(Float64, Irrational{:π})
```

```
tp, tp2 = typeof(pi_sum), typeof(pi_sum2)
```

```
println("While pi_sum is a $tp" * ", pi_sum2 is an $tp2")
```

```
# you can also print an object (useful when running simulations) using:
print(aps) # prints in the same line
```

```
println(aps) # prints on the next line (useful for printing in loops, etc)
```

Checking object types

As in R, Julia has a range of object types depending on the type of variable being stored:

```
Int64
```

```
# to check the type of any object use:
typeof(nu)
```

Note - Julia is like R and Python, it can infer the type of object (Integer, Float, etc) 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 = 3.141592
```

```
pi_sum1 = Float64(3.141592)
```

```
(Float64, Irrational{:π})
```

```
# you can then check the object type using:
typeof(pi_sum), typeof(pi_sum2) # note here that by using the preallocated variable pi, you are actually using an object of type Irrational (a specific type of Float)
```

Converting

It might also be useful to convert the type of an object from one type to another (when possible). This is done using the `convert` function.

```
a = 2
```

```
# allocate an Integer:
a = 2
```

```
b = 2.0
```

```
# convert to a Float:
b = convert(Float64, a)
```

```
b2 = 2.0
```

```
# or:
b2 = Float64(a)
```

Simple mathematical operations

```
c = 2
```

```
c = 2
```

```
d = 3
```

```
d = 3
```

```
sumcd = 5
```

```
# plus:
sumcd = c + d
```

```
diffcd = -1
```

```
# minus:
diffcd = c - d
```

```
productcd = 6
```

```
# multiplication:
productcd = c * d
```

```
divcd = 0.6666666666666666
```

```
# division:
divcd = c / d
```

```
powcd = 4
```

```
# exponent:
powcd = c^2
```

For a complete list of all mathematical operations see **the Julia manual**.

Arrays

Once you've mastered simple allocation, the next step is to create and store objects in arrays:

```
ar =  Int64[1, 2, 3, 4, 5]
```

```
# a one-dimensional array (or vector, a list of ordered data with a shared type) can be created using:
ar = [1,2,3,4,5] # square brackets act like c() in R
```

```
br =  String["Pint", "of", "moonshine", "please"]
```

```
# or:
br = ["Pint", "of", "moonshine", "please"]
```

Indexing

You can then access the different elements of an array by indexing:

```
1
```

```
# first position:
ar[1]
```

```
"moonshine"
```

```
# third position:
br[3]
```

```
Int64[2, 4]
```

```
# second and fourth postion:
ar[[2,4]]
```

```
String["moonshine", "please"]
```

```
# third and fourth position:
br[3:4]
```

Pre-allocation

You can also use some of the built in 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 = Float64[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
# create a vector of length 10 filled with 0's:
emptyvec = zeros(10)
```

```
onesvec = Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

```
# can also be done with ones:
onesvec = ones(10)
```

```
boolvec1 =
BitArray{1}: [false, false, false, false, false, false, false, false, false, fals
```

```
# or boolean values (TRUE or FALSE):
boolvec1 = falses(10)
```

```
boolvec2 =
BitArray{1}: [true, true, true, true, true, true, true, true, true, true]
```

```
boolvec2 = trues(10)
```

Empty arrays

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

```
I_array = Any[]
```

```
# empty array of any type:
I_array = []
```

```
J_array = Float16[]
```

```
# you can also justify the type if needed:
J_array = Float16[]
```

What's nice about Julia is that you don't have to provide an nrow or size argument like you would in R. This comes in very handy when looping and allocating. It is also worth noting that the I_array object you've just created behaves very similar to a `list()` in R and can handle many different data forms - for example, it can used to store n dimensional matrices or text.

```
range_array = 0.0:1.0:10.0
```

```
# sequence from 0-10 with length 11:
range_array = range(0, 10, length = 11)
```

```
range_array2 =   StepRange{Int64,Int64}[0:1:10]
```

```
# alternative:
range_array2 = [0:1:10]
```

```
StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}
```

```
typeof(range_array) # note that this produces an object of type StepRange and not a vector
```

```
range_collected =
  Float64[0.0,  1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0,  9.0,  10.0]
```

```
# you can turn range_array2 into a vector by "collecting":
range_collected = collect(range_array)
```

```
range_collected2 =   Int64[0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10]
```

```
# or you can use the ';' as a last argument in the [] to automatically collect:
range_collected2 = [0:1:10;]
```

```
Array{Int64,1}
```

```
# note that one of these methods produces an array of type Integer, the other of type Float:
typeof(range_collected2)
```

```
  Float64[0.0,  1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0,  9.0,  10.0]
```

```
# you can also convert the type of an array:
convert(Array{Float64,1}, range_collected2) # conversion from Int to Float
```

Both the `collect()` and `range()` are useful when setting up an experiment or looping over a variable of interest, as are the `length()` and `unique()` commands. Both `length()` and `unique()` operate the same way they do in R.

Broadcasting

To apply a built in function to all elements of a vector (or matrix), use the `.` operator. This is called broadcasting:

```
exp_array =
  Float64[1.0,  10.0,  100.0,  1000.0,  10000.0,  100000.0,  1.0e6,  1.0e7,  1.0e8,  1.0e9,
```

```
# map the exp10 function to all elements of range_array:
exp_array = exp10.(range_array)
```

```
log_array =
  Float64[-Inf,  0.0,  0.693147,  1.09861,  1.38629,  1.60944,  1.79176,  1.94591,  2.07944,
```

```
# map the log function:
log_array = log.(range_array)
```

Appending

You can also bind new elements to an array using the `append!` command:

```
dr =   Int64[1,  2,  3,  4,  5,  6,  7,  8,  9,  10]
```

```
# appending:
dr = append!(ar, 6:10)
```

Matrices

Matrices are created in a very similar way to arrays. In fact, it is easy to think of matrices as multi-dimensional arrays:

```
mat = 2×3 Array{Int64,2}:
       1  2  3
       4  5  6
```

```
# create a two-dimensional matrix:
mat = [1 2 3; 4 5 6] # rows are seperated by ; and columns by spaces
```

```
mat2 = 2×3×4 Array{Float64,3}:
       [:, :, 1] =
        0.0  0.0  0.0
        0.0  0.0  0.0

       [:, :, 2] =
        0.0  0.0  0.0
        0.0  0.0  0.0

       [:, :, 3] =
        0.0  0.0  0.0
        0.0  0.0  0.0

       [:, :, 4] =
        0.0  0.0  0.0
        0.0  0.0  0.0
```

```
# create a three-dimensional matrix filled with 0's:
mat2 = zeros(2,3,4) # number of rows, columns and dimensions
```

Elements of a matrix can also be accessed by indexing. As in R, rows are indexed first and columns second [row, column]: `

2

```
# first row of the second column:
mat[1,2]
```

```
Int64[3,  6]
```

```
# first two rows of the third column:
mat[1:2,3]
```

3

```
# if you provide 1 value:
mat[5] # it reads the matrix row-wise and then column-wise (hence mat[5] = 3 and not 5)
```

DataFrames and CSVs

Dataframes and CSVs are also easy to use in Julia and can help when storing, inputting and outputting data in a ready to use format.

Creating a dataframe

To initialise a dataframe you use the `DataFrame` function from the `DataFrames` package:

```
dat =
```

col1	col2	col3

```
# create a dataframe with three columns
dat = DataFrame(col1=[], col2=[], col3=[]) # as in section 1.7, we use [] to specify an empty column of any type and size
```

```
dat1 =
```

col1	col2	col3

```
# you can also specify column type using:
dat1 = DataFrame(col1=Float64[], col2=Int64[], col3=Float64)
```

```
dat2 =
```

species	size	rate

```
# and provide informative column titles using:
dat2 = DataFrame(species=[], size=[], rate=[])
```

Allocating to a dataframe

Once you've created a dataframe, you can allocate to it easily using the `push!` command:

```
x = 33.0
```

```
# allocation using push! command:
x = rand((3.5, 33.0))
```

```
y = 1
```

```
y = rand((1,7))
```

```
z = 100.0
```

```
z = rand((100.0, 700.00))
```

	col1	col2	col3
1	33.0	1.0	100.0

```
push!(dat, [x,y,z])
```

```
species = "Atlantic cod"
```

```
# or
species = "Atlantic cod"
```

```
size = 86
```

```
size = 86
```

```
rate = 3.0
```

```
rate = 3.0
```

	species	size	rate
1	"Atlantic cod"	86	3.0

```
push!(dat2, [species, size, rate])
```

Exploring a dataframe

```
x2 = 55.6
```

```
# add a second row to dat
x2 = rand((55.6, 77.1))
```

```
y2 = 9
```

```
y2 = rand((9,11))
```

```
z2 = 80.0
```

```
z2 = rand((10.0, 80.00))
```

	col1	col2	col3
1	33.0	1.0	100.0
2	33.0	1.0	100.0

```
push!(dat, [x,y,z])
```

```
# look at a dataframe:
print(dat2)
```

	col1	col2	col3
1	33.0	1.0	100.0

```
# first row:
first(dat,1)
```

DataFrameRow (3 columns)

	col1	col2	col3
	Any	Any	Any
2	33.0	1.0	100.0

```
# last row:
last(dat)
```

DataFrameRow (3 columns)

	col1	col2	col3
	Any	Any	Any
1	33.0	1.0	100.0

```
# you can also view or extract all rows or all columns using:
dat[1,:] # : all columns
```

```
Any[33.0,  33.0]
```

```
dat[:,1] # : all rows
```

```
Any["Atlantic cod"]
```

```
# alternatively, you can select columns using their names:
dat2[:species]
```

CSV files

Dataframes can be written out (stored/saved) into your active project directory as .csv files using the `CSV` package:

```
"myfishdata.csv"
```

```
# write out a CSV:
CSV.write("myfishdata.csv", dat2)
```

```
dat_in =
```

	species	size	rate
1	"Atlantic cod"	86	3.0

```
# read in a CSV:
dat_in = CSV.read("myfishdata.csv", DataFrame)
```

```
# alternatively, files can be exported as text files:
writedlm(join(["testing_", dat2[1,1], "_rates.txt"]), mat, "\t") # join does what it says on the tin - it joins variables into a single string
```

```
dat_txt = 2×3 Array{Float64,2}:
           1.0  2.0  3.0
           4.0  5.0  6.0
```

```
# or imported as text files:
dat_txt = readdlm(join(["testing_", dat2[1,1], "_rates.txt"]))
```

Usually, we tend to use CSV's for dataframes and delimited files for matrices but there is no hard and fast rule.

We also recommend naming your files as something memorable and informative. When you're running a experiment and outputing thousands of files, a future you will thank you for naming your files with care. If outputing many files, we would also recommend creating seperate folders for your output files. Folders are easier to read back into Julia or R and no one likes a cluttered project directory.

Functions

Functions work exactly like they do in R, however, there are three fundamental differences:

- there is no need for `{}` brackets (thank god)
- indenting (Julia requires seperate parts of a function to be indented - don't worry, VS Code should do this for you)
- scoping (we'll attempt to explain this later)

Functions start with the word `function` and end with the word `end`. To store something that is calculated in a function, you use the `return` command.

```
plus_two (generic function with 1 method)
```

```
# simple function:
function plus_two(x)
    return x + 2
end
```

```
x3 = 17
```

```
# input variable:
x3 = 17
```

```
z3 = 35.0
```

```
# run functions:
z3 = plus_two(x)
```

```
pub_time (generic function with 1 method)
```

```
# functions can also be witten to take no inputs:
function pub_time()
    println("Surely it's time for Interval")
    return
end
```

```
pub_time()
```

Positional arguments

Unlike in R, input variables for functions have to be specified in a fixed order unless they have a default value which is explictly specified. For instance, we can build a function that measures body weight on different planets:

```
bodyweight (generic function with 2 methods)
```

```
# weight function:
function bodyweight(BW_earth, g = 9.81)
    return BW_earth*g/9.81
end
```

```
80.0
```

```
# if you execute:
bodyweight(80)
```

And get your weight on another planet

```
md"And get your weight on another planet"
```

```
30.33639143730887
```

```
# and don't specify g, you get body weight as measured on Earth (because g is fixed at a default value of 9.81)
# alternatively, you can change g
bodyweight(80, 3.72)
```

Keyword arguments

In Julia, you can't change the order of input variables for a function or circumvent the problem by specifing the name of an input variable (like you can in R or Python). To overcome this, you can use keyword arguments which have no fixed position:

```
key_word (generic function with 2 methods)
```

```
# function with keyword arguments:
function key_word(a, b=2; c, d=2) # here, b and d are fixed, a is positional and c is a keyword argument
    return a + b + c + d
end
```

```
8
```

```
# the addition of ; before c means that c is an keyword argument and can be specified in any order
# however, you do have to specify it by name:
key_word(1, c=3)
```

```
16
```

```
# or:
key_word(1, 6, c=7)
```

```
UndefKeywordError: keyword argument c not assigned

  1. top-level scope @ Local: 2 [inlined]
```

```
# if you don't provide a c argument, Julia will return an error:
key_word(1, 8, d=4)
```

keyword arguments must always be specified

Loops

Loops are useful tools in any programming language. Loops allow you to iterate over elements of a vector or matrix and calculate things of interest.

For loops

For loops work by iterating over a specified range (e.g. 1-10) at specified intervals (e.g. 1,2,3...). For instance, we might use a for loop to fill an array:

```
I_array2 =   Any[]
```

```
# create some arrays:
I_array2 = []
```

```
tab =   Any[]
```

```
tab = []
```

```
# for loop to fill an array:
for i in 1:1000
    for_test = rand((1,2)) # pick from the number 1 or 2 at random
    push!(I_array2, for_test) # push! and store for_test in I_array2 - Julia is smart enough to do this iteratively, you don't neccessarily have to indexed by `[i]` like you might do in R
end
```

```
Any[2, 1, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1,    more , 1
```

```
I_array2
```

```
# nested for loop to fill an array:
for k in 1:4
    for j in 1:3
        for i in 1:2
            append!(tab,[[i,j,k]]) # here we've use append! to allocate iteratively to the array as opposed to using push! - both work.
        end
    end
end
```

```
Any[Int64[1, 1, 1], Int64[2, 1, 1], Int64[1, 2, 1], Int64[2, 2, 1], Int64[1, 3,
```

```
tab
```

```
table = 2×3×4 Array{Float64,3}:
[:, :, 1] =
 0.0  0.0  0.0
 0.0  0.0  0.0

[:, :, 2] =
 0.0  0.0  0.0
 0.0  0.0  0.0
```

```
[:, :, 3] =
 0.0  0.0  0.0
 0.0  0.0  0.0

[:, :, 4] =
 0.0  0.0  0.0
 0.0  0.0  0.0
```

```
# you can also use for loops to allocate to a matrix:
table = zeros(2,3,4) # [2,3,4] matrix
```

```
for k in 1:4
    for j in 1:3
        for i in 1:2
            table[i,j,k] = i*j*k # allocate i to rows, j to columns and k to dimensions
        end
    end
end
```

```
2×3×4 Array{Float64,3}:
[:, :, 1] =
 1.0  2.0  3.0
 2.0  4.0  6.0

[:, :, 2] =
 2.0  4.0   6.0
 4.0  8.0  12.0

[:, :, 3] =
 3.0   6.0   9.0
 6.0  12.0  18.0

[:, :, 4] =
 4.0   8.0  12.0
 8.0  16.0  24.0
```

```
table
```

```
persons =  String["Alice", "Alice", "Bob", "Bob2", "Carl", "Dan"]
```

```
# or play around with strings:
persons = ["Alice", "Alice", "Bob", "Bob2", "Carl", "Dan"]
```

```
for person in unique(persons)
    println("Hello $person")
end
```

There are tons of different functions that can be helpful when building loops. Take a few minutes to look into `eachindex`, `eachcol`, `eachrow` and `enumerate`. They all provide slightly different ways of telling Julia how you want to loop over a problem. Also, remember that loops aren't just for allocation, they can also be very useful when doing calculations.

If, else and break

When building a loop, it is often meaningful to stop or modify the looping process when a certain condition is met. For example, we can use the `break`, `if` and `else` statements to stop a for loop when `i` exceeds 10:

```
# if and break:
for i in 1:100
    println(i) # print i
```

```
    if i >10
        break # stop the loop with i >10
    end
end
```

```
# this loop can be modified using an if-else statement:
for j in 1:100
    println(j) # print i
    if j >10
        break # stop the loop with i >10
    else
        println(j^3)
    end
end
```

You'll notice that every statement requires it's own start and `end` points, and is indented as per Julia's requirements. `if` and `else` statements can be very useful when building experiments, for example we might want to stop simulating a network `if` more than 50% of the species have gone extinct.

Continue

The `continue` command is the opposite to `break` and can be useful when you want to skip an iteration but not stop the loop:

```
for i in 1:30
    if i % 3 == 0
        continue # makes the loop skip iterations that are a multiple of 3
    else println(i)
    end
end
```

While

`While` loops provide an alternative to `for` loops and allow you to iterate until a certain condition is met:

```
begin #this is not needed in the REPL/VScode/Atom et al
    global i=0 # set a counter in the global scope
    while(i<30) # justify a condition
        println(i) # prints i until i < 30
        global i += 1 # count
    end
end
```

While loops don't require you to specify a looping sequence (e.g. `i in 1:100`). This can be very useful because sometimes you simply don't know how many iterations you might need.

Plots

In R, the plotting of data is either done in base R or via the `ggplot2` package. If you're a base R person, you'll probably feel more comfortable with the `Plots` package. Alternatively, if you prefer

`ggplot2`, the `Gadfly` package is the closest thing you'll find in Julia. We'll introduce both in the following sections.

It is worth noting that Julia is based on a 'Just in Time' compiler (or JIT) so the first time you call a function it needs to compile, and can take longer than expected. This is especially true when rendering a plot. Consequently, the first plot you make might take some time but it gets significantly faster after that.

Plots

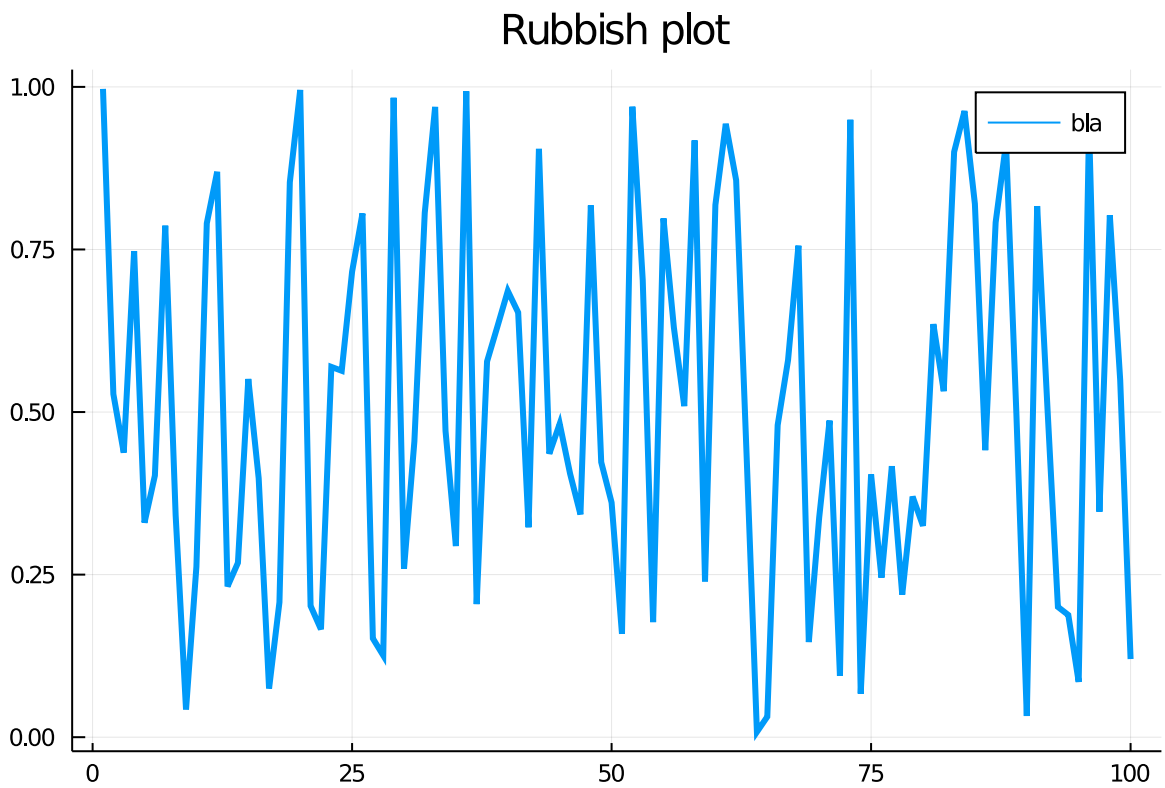
Making a basic plot is pretty straightforward in Julia:

```
x4 = 1:100
```

```
# basic plot:
x4 = 1:100
```

```
y4 =
  Float64[0.996921, 0.527617, 0.437537, 0.747464, 0.329862, 0.402339, 0.786622, 0.341
```

```
y4 = rand(100)
```




```
# the plot will open in a new tab:
Plots.plot(collect(x4),y4,label="bla", title = "Rubbish plot", lw = 3)
```

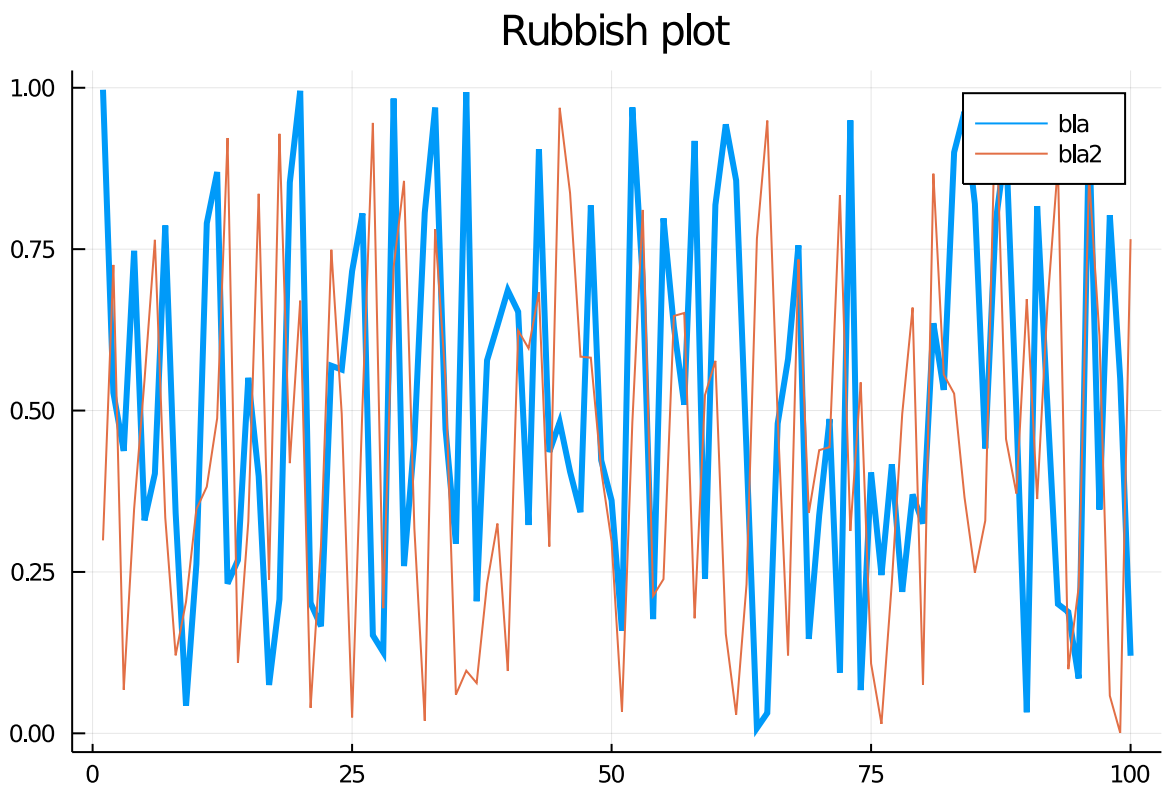
The `Plots.plot()` statement tells Julia that you want to use the `plot` function within the `Plots` package. We're using it here as the `Gadfly` package is also active and has it's own `plot` function.

To mutate a plot use `!`:

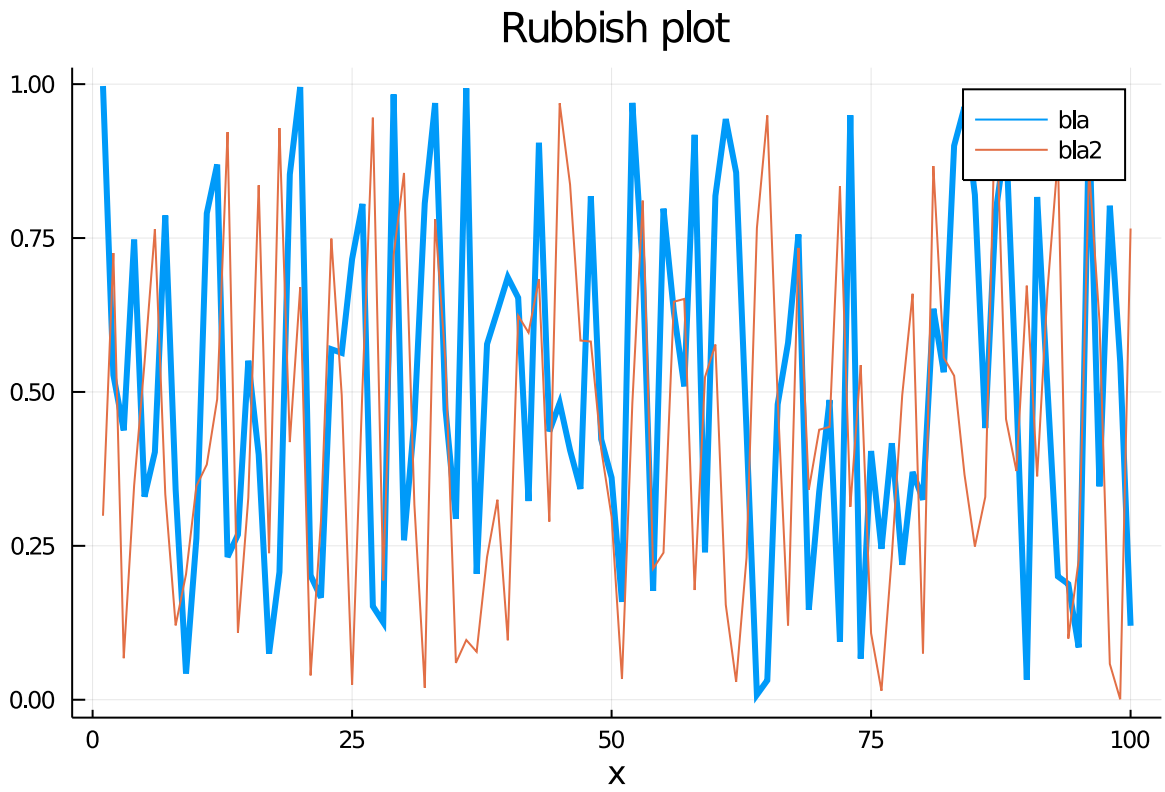
```
z4 =
```

```
Float64[0.298894, 0.725595, 0.0674993, 0.347755, 0.547088, 0.764432, 0.334162, 0.12
```

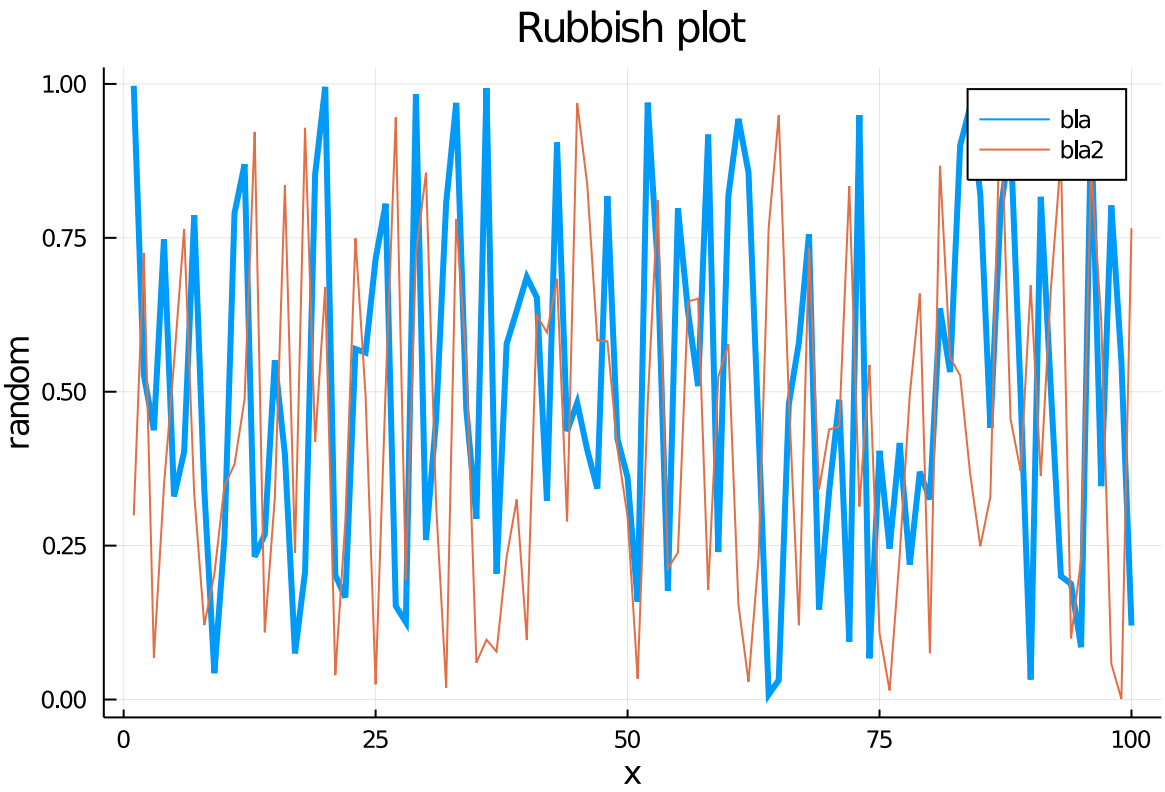
```
z4 = rand(100)
```




```
# add a second line to your plot:
plot!(x4,z4,label="bla2")
```

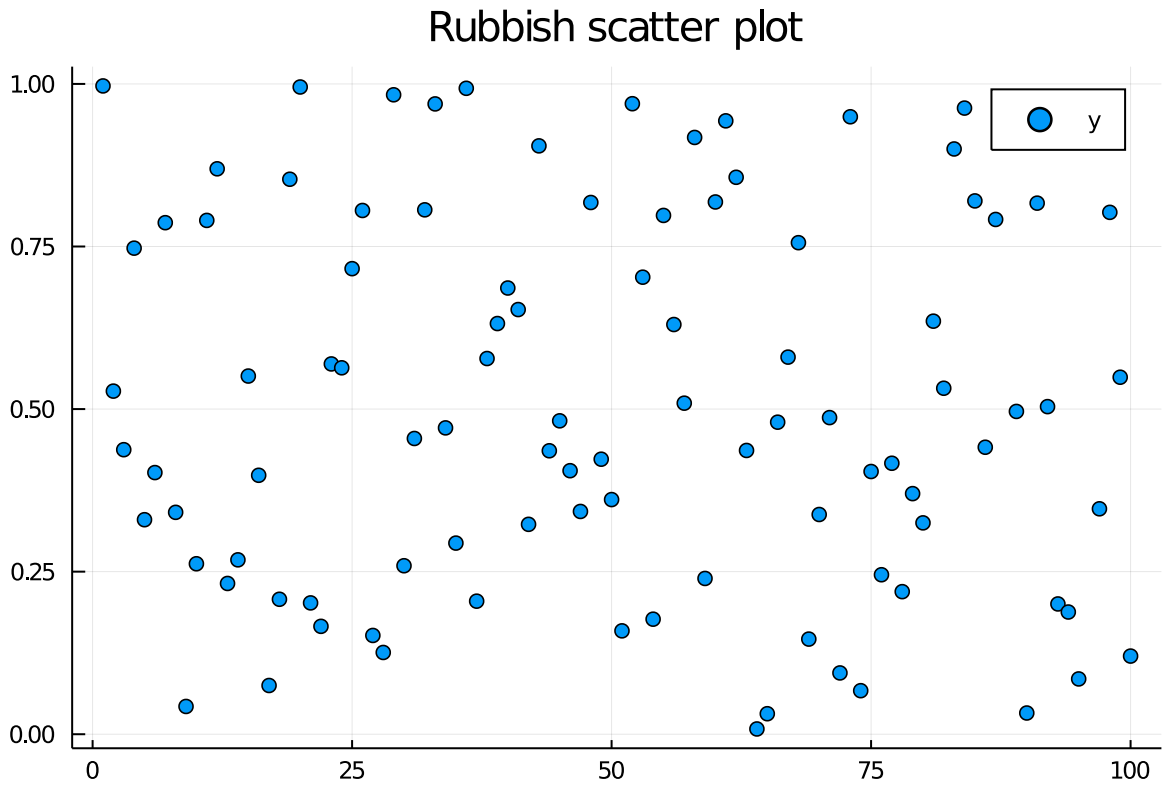



```
# adds an x-axis label:
xlabel!("x")
```

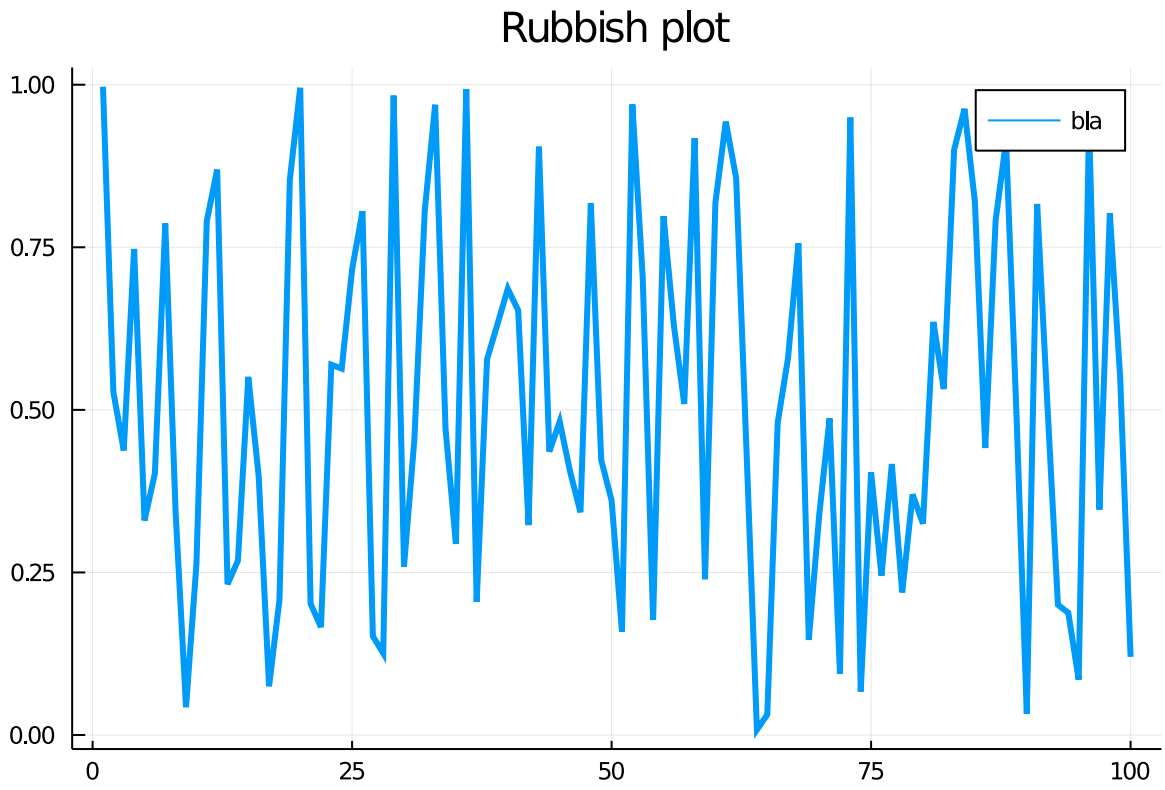



```
# and y-axis label:
ylabel!("random")
```

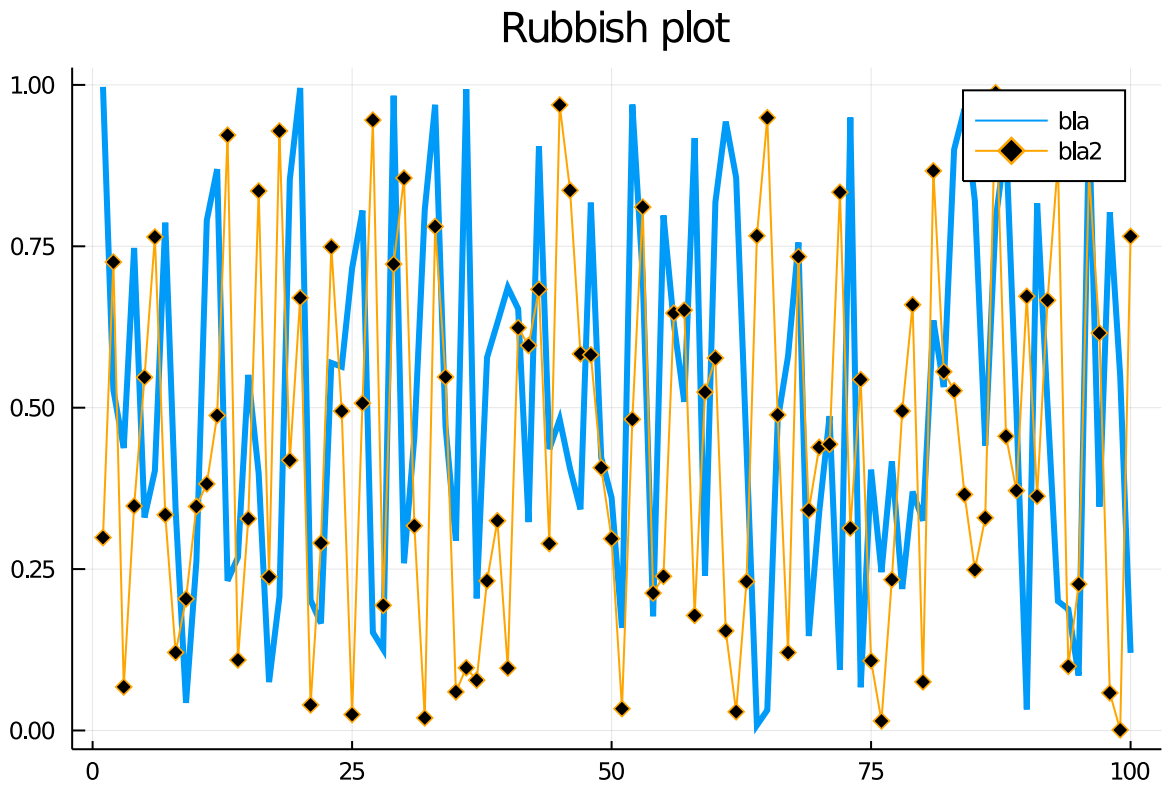
You can change the plot type using the `seriestype` argument or by using a built in plot macro (e.g. `scatter`):



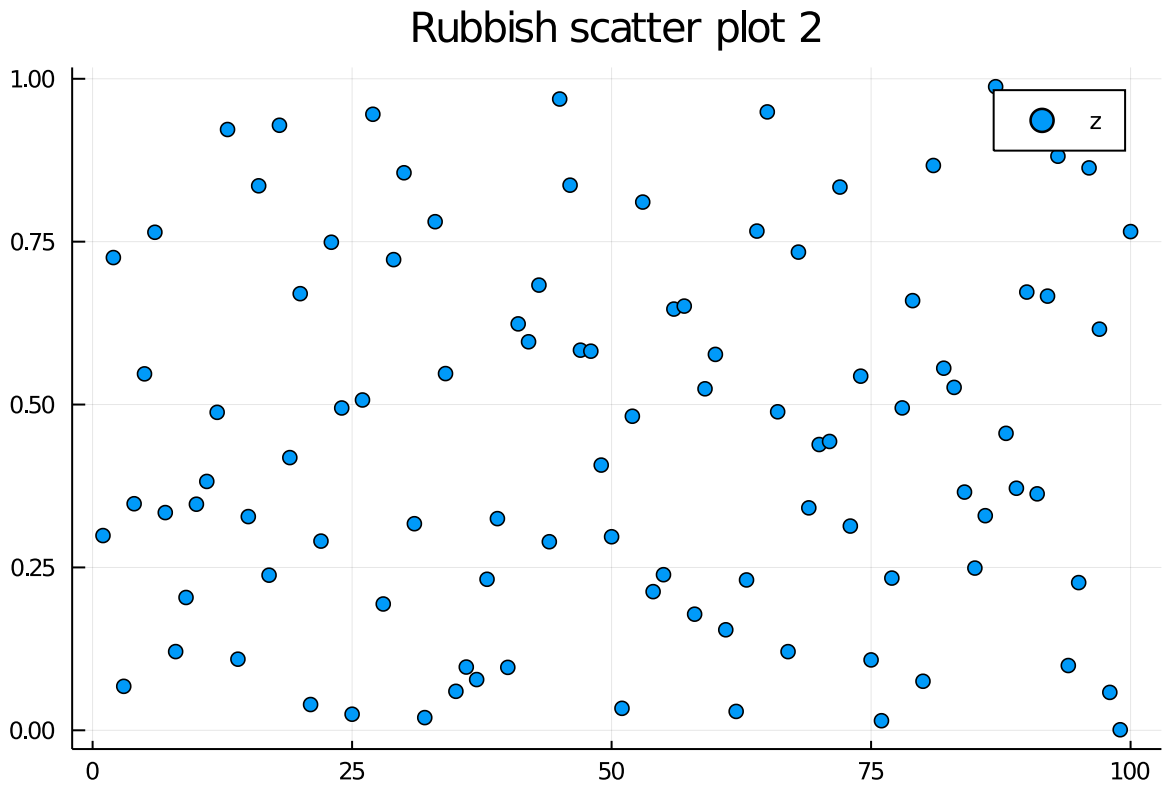

```
# seriestype:
Plots.plot(x4,y4, title = "Rubbish scatter plot", seriestype = :scatter, label =
"y")
```




```
# or:
Plots.plot(x4,y4,label="bla", title = "Rubbish plot", lw = 3) # new plot
```



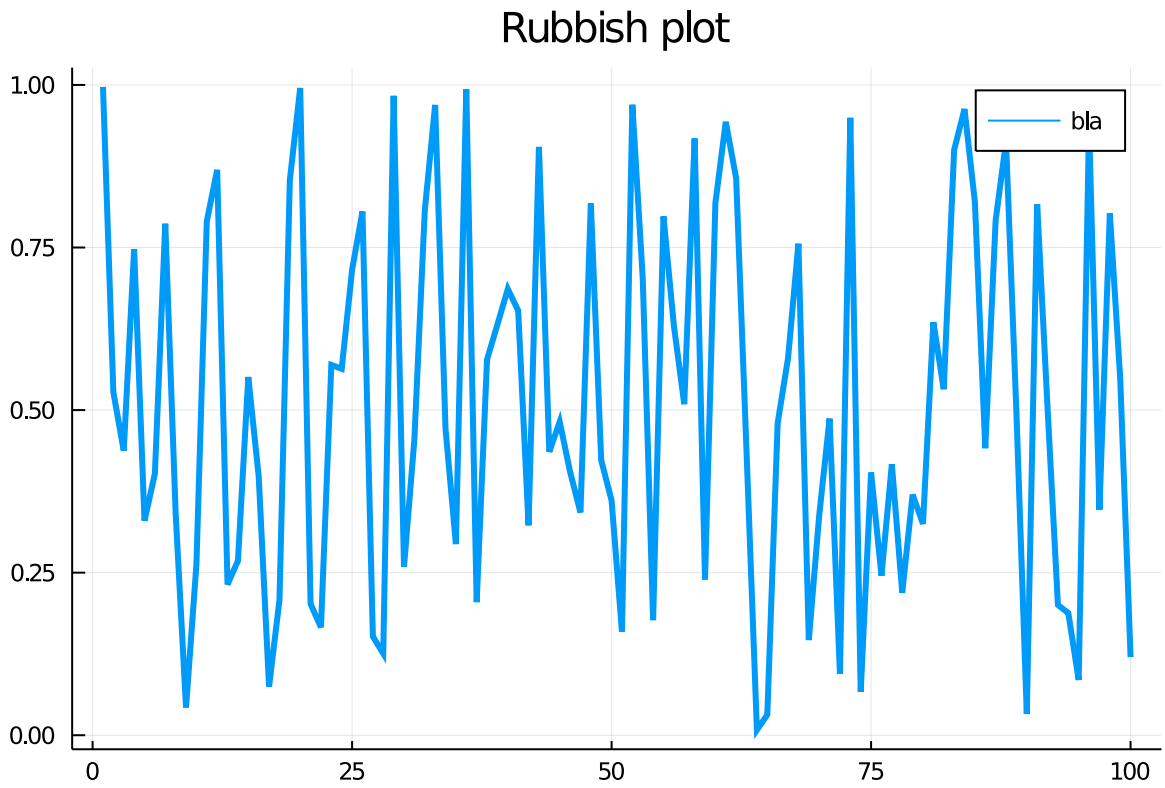

```
plot!(z4, seriestype = [:line :scatter], lc = :orange, mc = :black, msc = :orange,
label = "bla2", markershape = :diamond)
```




```
# scatter macro:
scatter(x4, z4, title = "Rubbish scatter plot 2", label = "z")
```

`mc` is for marker colour and `lc` for line color.

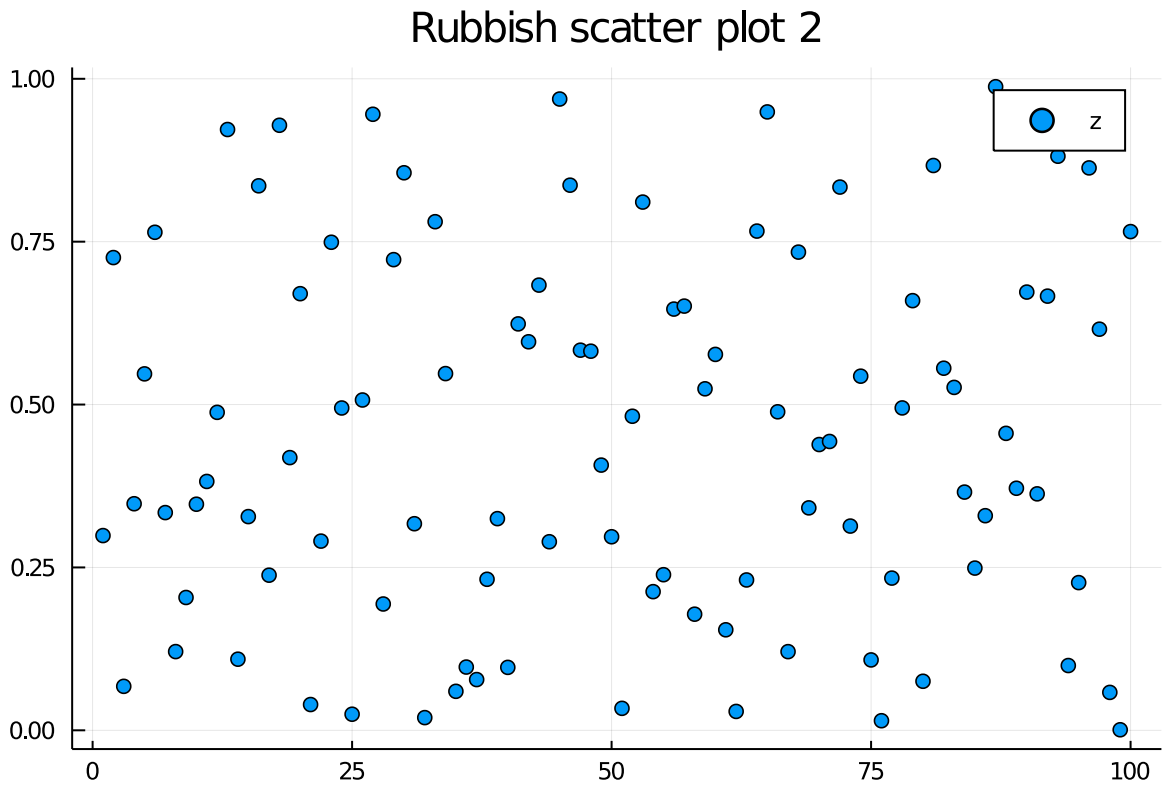
Plots can be saved and outputted using `savefig` or by using an output marco (e.g. `png`)




```
Plots.plot(x4,y4,label="bla", title = "Rubbish plot", lw = 3)
```

```
# savefig saves the most recent plot:
savefig("plot.png")
```

```
p1 =
```




```
# here the file type is explicitly stated
# you could also the macro:
p1 = scatter(x4, z4, title = "Rubbish scatter plot 2", label = "z")
```

```
png(p1,"plot2")
```

Once you've created a plot it can be viewed or reopened in VS Code by navigating to the `Julia explorer: Julia workspace` symbol in the activity bar (three circles) and clicking on the plot object. We advise that you always name and assign your plots (e.g. p1, p2, etc). The `Plots` package also has it's own **tutorial** for plotting in Julia.

Gadfly

You can also plot in Julia using the `Gadfly` package. `Gadfly` can be especially useful when working with dataframes. The documentation for the `Gadfly` package can be found **here**.

Let's start by querying an online dataset:

```
crabs =
```

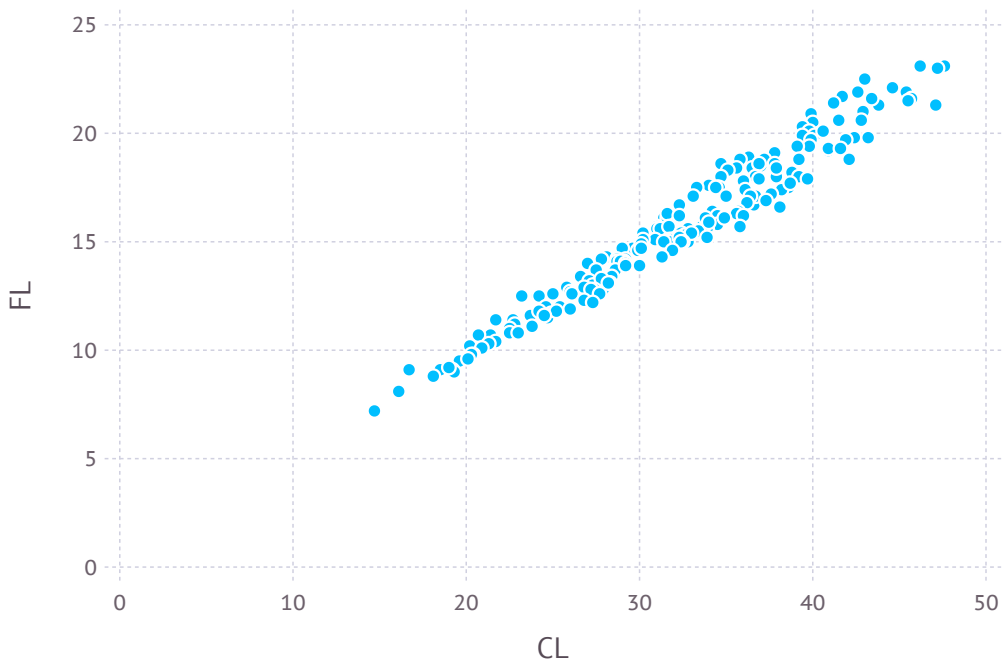
	Sp	Sex	Index
1	CategoricalValue{String,UInt8} "B"	CategoricalValue{String,UInt8} "M"	1
2	CategoricalValue{String,UInt8} "B"	CategoricalValue{String,UInt8} "M"	2
3	CategoricalValue{String,UInt8} "B"	CategoricalValue{String,UInt8} "M"	3
4	CategoricalValue{String,UInt8} "B"	CategoricalValue{String,UInt8} "M"	4
5	CategoricalValue{String,UInt8} "B"	CategoricalValue{String,UInt8} "M"	5

	Sp	Sex	Index
6	CategoricalValue{String,UInt8} "B"	CategoricalValue{String,UInt8} "M"	6
7	CategoricalValue{String,UInt8} "B"	CategoricalValue{String,UInt8} "M"	7
8	CategoricalValue{String,UInt8} "B"	CategoricalValue{String,UInt8} "M"	8

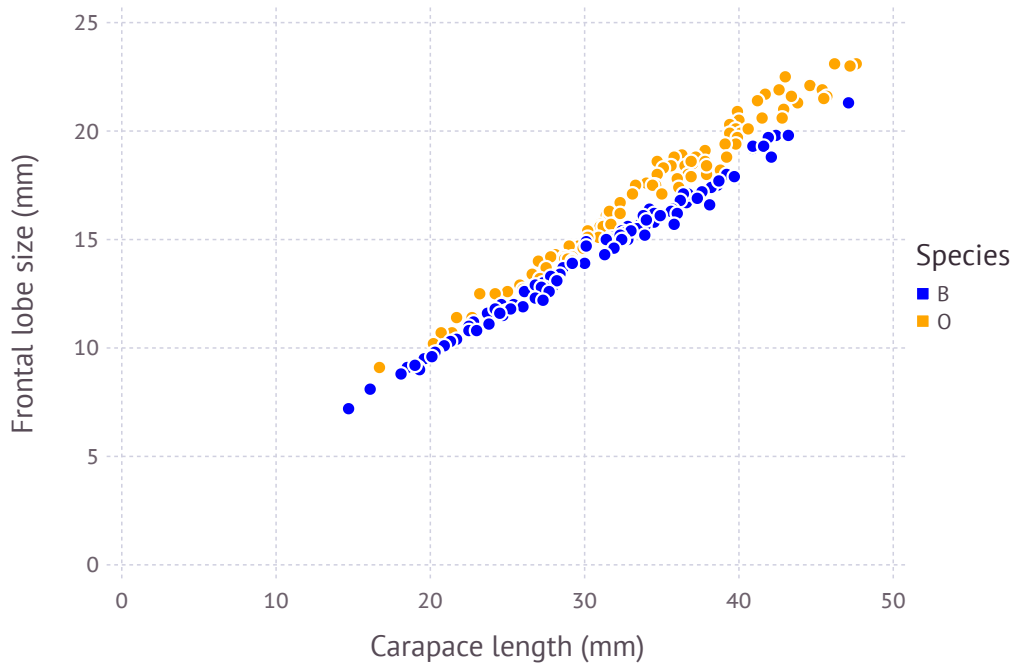
```
# data query from https://stat.ethz.ch/R-manual/R-devel/library/MASS/html/crabs.html
crabs = dataset("MASS", "crabs")
```

	variable	mean	min	median	max
1	:Sp	nothing	CategoricalValue{String,UInt8} "B"	nothing	CategoricalValue{Str
2	:Sex	nothing	CategoricalValue{String,UInt8} "F"	nothing	CategoricalValue{Str
3	:Index	25.5	1	25.5	50
4	:FL	15.583	7.2	15.55	23.1
5	:RW	12.7385	6.5	12.8	20.2
6	:CL	32.1055	14.7	32.1	47.6
7	:CW	36.4145	17.1	36.8	54.6
8	:BD	14.0305	6.1	13.9	21.6

```
# explore data:
describe(crabs) # gives a quick decription of the data
```




```
# plot crab carapace length by body depth:
Gadfly.plot(crabs, x = :CL, y = :FL)
```




```
# colour by species:
Gadfly.plot(
    crabs, x = :CL, y = :FL
    , color = :Sp
    , Guide.xlabel("Carapace length (mm)")
    , Guide.ylabel("Frontal lobe size (mm)")
    , Guide.colorkey(title="Species")
    , Scale.color_discrete_manual("blue", "orange")
    )
```

B = blue crab and O = orange crab

Again, we've used the `Gadfly.plot()` function as both the `Plots` and `Gadfly` packages are active. If only one was active, we could just use `plot()`.

Scoping

Scoping refers to the accessibility of a variable within your project. The scope of a variable is defined as the region of code where a variable is known and accessible. A variable can be in the `global` or `local` scope.

Global

A variable in the `global` scope is accessible everywhere and can be modified by any part of your code. When you create (or allocate to) a variable in your script outside of a function or loop you're creating something that is `global`:

```
A = 7
```

```
# global:
A = 7
```

```
B =
```

```
OffsetArray(::Array{Float64,1}, 1:10): [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
```

```
# global array:
B = zeros(1:10)
```

```
7
```

```
# you can also force a variable to be global using the global macro:
global C = 7
```

Local

A variable in the `local` scope is only accessible in that scope or in scopes eventually defined inside it. When you define a variable within a function or loop that isn't returned then you create something that is `local`:

```
C2 =  Float64[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
C2 = zeros(10) #pre-allocate
```

```
# local:
for i in 1:10
    local_varb = 2 # local_varb is defined inside the loop and is therefore local (only accessible within the loop)
    C2[i] = local_varb*i # in comparison, C is defined outside of the loop and is therefore global
end
```

```
UndefVarError: local_a not defined
```

1. top-level scope @ *Local: 1*

```
local_a # returns a 'not defined' error
```

```
Float64[2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0]
```

```
C2 # returns a vector
```

Quick tips

Some quick tips that we've learnt the hard way...

1. In the REPL, you can use the up arrow to scroll through past code
2. You can even filter through your past code by typing the first letter of a line previously executed in the REPL. For example, try typing p in the REPL and using the up arrow to scroll through your code history, you should quickly find the last plot command you executed.
3. Toggle word wrap via View>Toggle Word Wrap or alt-Z
4. Red wavy line under code in your script = error in code
5. Blue wavy line under code in your script = possible error in code
6. Errors and possible erros can be viewed in the `PROBLEMS` section of the REPL
7. You can view your current variables (similar to the top right hand panel in RStudio) by clickling on the `Julia explorer: Julia workspace` symbol in the activity bar (three circles). You can then look at them in more detail by clicking the sideways arrow (when allowed).

8. Julia has a strange copying aspect where if `a=b`, any change in `b` will automatically cause the same change in `a`. For example:

```
aa =  Int64[1, 2, 3]
```

```
aa = [1,2,3]
```

```
bb =  Int64[1, 2, 3]
```

```
bb = aa
```

```
print(bb)
```

```
41
```

```
bb[2] = 41
```

```
Int64[1, 41, 3]
```

```
aa
```

This approach is advantageous because it lets Julia save memory, however, it is not ideal. As a result we might want to force `c` to be an independent copy of `a` using the `deepcopy` function:

```
cc =  Int64[1, 41, 3]
```

```
cc = deepcopy(aa)
```

```
101
```

```
cc[3] = 101
```

```
Int64[1, 41, 101]
```

```
cc
```

```
Int64[1, 41, 3]
```

```
aa
```

9. You can view a .csv or .txt file by clicking on a file name in the project directory (left panel) - this opens a viewing window. CSV's also have a built in 'Preview' mode - try using right click>Open Preview on a .csv file and check it out.