

Scientific Programming in Julia

Statistics, Functional Programming, and Performance Computing

Nicholas Gale and Stephen Eglen

DataFrames

- ▶ Dataframes are tabular data with named columns.
- ▶ They are supported using the DataFrames package.
- ▶ The names are given in symbols (`:xyz` which is a name, differing from the variable `xyz`).

Creating DataFrames

- ▶ To initialise a data frame use the `dataframe(colname1=data1, colname2=data2)`
- ▶ No recycling – columns must have matching lengths.
- ▶ An empty data frame is made from empty typed-vectors.

```
using DataFrames
```

```
println(DataFrame(x=[0.5, 6, 7],  
                  y=[3, 5, 9]))
```

3×2 DataFrame

Row	x	y
	Float64	Int64
1	0.5	3
2	6.0	5
3	7.0	9

```
println(  
  DataFrame(x=Float64[],  
            st=String[]))
```

0×2 DataFrame

Row	x	st
	Float64	String

Accessing DataFrames

- ▶ Data indexed as a matrix, by rows, cols and symbols.
- ▶ Entire rows can be indexed as normal with : or by !.

```
df = DataFrame(A=[5, 6, 7],  
               B='a':'c',  
               C=["cat", "dog",  
                 "eel"],  
               )  
println(df)
```

3x3 DataFrame

Row	A	B	C
	Int64	Char	String
1	5	a	cat
2	6	b	dog
3	7	c	eel

```
println(df[!, :A])  
println(df[2,3])  
println(df[:, [:C, :A]])
```

[5, 6, 7]

dog

3x2 DataFrame

Row	C	A
	String	Int64
1	cat	5
2	dog	6
3	eel	7

Modifying DataFrames

- ▶ The `push!` function is used to add more rows. The function accepts an ordered tuple.
- ▶ `promote=true` needed if symbol's datatype does not match the data.

```
df = DataFrame(S=String[], N=Float64[])  
push!(df, ("A string", 4.4))  
# println(push!(df, ('C', 4.4))) # WRONG - C is char  
println(push!(df, ('C', 4.4), promote=true))
```

2x2 DataFrame

Row	S	N
	Any	Float64
1	A string	4.4
2	C	4.4

CSV Piping

- ▶ A common usecase for datascience is importing CSV data.
- ▶ The DataFrame function can be wrapped around a CSV file; or the file piped with `▷. (| >)`
- ▶ A DataFrame may be written to CSV using `CSV.write`.

```
using CSV
df1 = DataFrame(CSV.File("example.csv"))
df2 = CSV.File("example.csv") ▷ DataFrame
df1 == df2
```

true

Random Number Generation

- ▶ The default random function is `rand(...)` and is highly extensible.
- ▶ `rand(distribution, n)` takes `n` samples from a distribution.
- ▶ The default distribution is $U([0, 1])$ and default sample number is one. Distributions can be specified as a set.

```
using Random
rn = round( rand(), digits=6)
rsamp = round.( rand(3); digits=4)
rsampcustom = rand([1,4,"A"], 5)
display([rn, rsamp, rsampcustom])
```

3-element Vector{Any}:

0.862677

[0.8202, 0.6232, 0.8901]

Any[1, "A", "A", 4, 1]

Seeding and Permutation

- ▶ A seed is specified with: `Random.seed!(seed_number)`.
- ▶ A random permutation is given by the `randperm` function.

```
using Random
Random.seed!(1)
println(randperm(5))
Random.seed!(1)
println(randperm(5))
```

[4, 3, 5, 2, 1]

[4, 3, 5, 2, 1]

Statistics

- ▶ Julia has first class support for statistics.
- ▶ The StatsBase package has the standard statistics functions: mean, var, std, mode, zscore, quantile et c.
- ▶ Weighted statistics are computed with an optional weights vectors; in R they are their own methods.

```
using StatsBase
x = [20, 0, 2, 4, 4]
w = Weights([1, 5, 5, 5, 5])
@show mean(x)
@show mean(x, w)
@show median(x);
```

```
mean(x) = 6.0
```

```
mean(x, w) = 3.3333333333333335
```

```
median(x) = 4.0
```

Distributions

- ▶ Julia supports distributions through Distributions package.
- ▶ Distribution fitting is provided through `fit(DistributionType, data)`; the result of which can then be sampled from.

```
using Distributions
data = 4*randn(1000) .+ 12
d = fit(Normal, data)
println(d)
rand(d, 4) # draw 4 samples
```

Normal{Float64}($\mu=12.042025684837661$, $\sigma=3.970852076846705$)

4-element Vector{Float64}:

11.52085194303134

9.64280094961792

3.2677297936107164

15.568757490372628

Sampling

- ▶ `sample` can be used as alternative to `rand`.
- ▶ `sample` needs a distribution (can be a categorical vector). Can be weighted.
- ▶ An optional keyword `replace=false` (default: `true`) can specify sampling without replacement.

```
using StatsBase
catdist = [1,4,5,10]
@show sample(catdist, 7)
@show sample(catdist, 3, replace=false);
# println(sample(catdist, 7, replace=false)) # WRONG!
```

```
sample(catdist, 7) = [1, 4, 4, 10, 10, 10, 1]
sample(catdist, 3, replace = false) = [4, 10, 1]
```

Interlude – special values

- ▶ missing is like NA in R:
<https://docs.julialang.org/en/v1/manual/missing/>
- ▶ Inf and -Inf available.
- ▶ NaN is available as part of IEEE standard, e.g. `var([1])`
- ▶ Constants like π and e available.

```
@show e^(im) $\pi$   $\approx$  -1
```

```
@show 1//10 + 2//10 == 3//10;
```

```
 $e$  ^ (im *  $\pi$ )  $\approx$  -1 = true
```

```
1 // 10 + 2 // 10 == 3 // 10 = true
```

Statistics Ecosystem

- ▶ The statistics ecosystem is large and well-supported. R is still number 1 for statistics though.
- ▶ Useful packages: StatsBase, Statistics, Distributions, DataFrames, HypothesisTests.
- ▶ Useful resource: <https://juliastats.org/>

Functional Programming

- ▶ Functional programming is a style where functions and data are cleanly separated.
- ▶ Object oriented programming is where data and functions/methods are attached to objects.
- ▶ Julia lends itself towards functional programming.

Map

- ▶ `map()` takes a function and applies it to an iterable: vector, range, etc.
- ▶ The function can be a function name, or an anonymous function.
- ▶ Multiple iterables can be passed for functions of multiple variables.

```
v = [0,  $\pi/6$ ,  $\pi/4$ ]  
sin1 = map(sin, v)  
autod_cos = map(x  $\rightarrow$  (cos(x), -sin(x)), v)  
mask = [0 1 0]  
bad_sin = map((x,y) $\rightarrow$ (y==1 ? sin(x) : missing), v, mask)  
display(autod_cos)  
println(bad_sin)
```

3-element Vector{Tuple{Float64, Float64}}:

(1.0, -0.0)

(0.8660254037844387, -0.49999999999999994)

(0.7071067811865476, -0.7071067811865475)

Union{Missing, Float64}[missing, 0.49999999999999994, missing]

Filter

- ▶ filter evaluates a logical condition over an iterable.
- ▶ filter! is an in-place operation; filter creates a new copy.

```
a = collect(1:10)
```

```
@show a
```

```
b = filter(iseven, a)
```

```
@show b
```

```
filter!(isodd, a)
```

```
@show a;
```

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

```
b = [2, 4, 6, 8, 10]
```

```
a = [1, 3, 5, 7, 9]
```

Sum

- ▶ The sum function supports functions as the first argument; these are applied before summing.
- ▶ The dimension/s which the sum is performed along is given by the `dims=(dim1,...)` keyword. Default is all.

```
m = hcat(collect(1:4), collect(5:8))  
display(m)  
sum(x → x^2, m, dims=2)
```

4×2 Matrix{Int64}:

1	5
2	6
3	7
4	8

4×1 Matrix{Int64}:

26
40
58
80

Reduce

- ▶ The reduce function behaves exactly the same way as in R.
- ▶ Can support a generic binary operation that can be distributed over an iterable.

```
v = [7, 9, 4, 8]
println( reduce( (x,y) → x < y ? x : y, v))
println( reduce( (x,y) → x < y ? x : y, v, init=2))
```

4

2

Mapreduce

- ▶ Common paradigm: map function f onto items, and then reduce items using binary operator.
- ▶ Saves on memory allocation compared to do both operations.

```
a = reshape(collect(1:12), (3,4))  
display(a)  
println( mapreduce(isodd, +, a, dims=1) )  
println( mapreduce(isodd, +, a, dims=2) )  
println( mapreduce(isodd, +, a) )
```

3×4 Matrix{Int64}:

1	4	7	10
2	5	8	11
3	6	9	12

[2 1 2 1]

[2; 2; 2;;]

6

Dimensions...

- Warning: R and Julia have switched the numerical code for rows and columns. (Julia is probably more consistent).

```
A = [1 2 3; 4 5 6]
display(A)
@show sum(A, dims=1)
@show sum(A, dims=2);
```

2×3 Matrix{Int64}:

1 2 3

4 5 6

sum(A, dims = 1) = [5 7 9]

sum(A, dims = 2) = [6; 15;;]

```
using RCall
@rput A
@R_str("apply(A, 1, sum)")
```

RObject{IntSxp}

[1] 6 15

eachrow() and eachcol()

- ▶ Julia's version of `apply(A, fn, dim)` is to use `eachrow` and `eachcol`.

```
A = reshape(collect(1:6), (2,3))  
@show A  
@show map(sum, eachrow(A))  
@show map(sum, eachcol(A))
```

```
A = [1 3 5; 2 4 6]  
map(sum, eachrow(A)) = [9, 12]  
map(sum, eachcol(A)) = [3, 7, 11]
```

3-element Vector{Int64}:

```
3  
7  
11
```

Inner and Outer product

- ▶ Inner product calculated with `dot(x, y)` or $x \cdot y$
- ▶ Unlike R's `outer()`, Julia has no *specific* method; but broadcasting a column vector to a row vector creates a matrix.

```
using LinearAlgebra
```

```
v = [1, 4, 2]
```

```
@show v·v
```

```
f(x, y) = x+y
```

```
@show f(v, v)
```

```
f.(v, v')
```

```
v · v = 21
```

```
f(v, v) = [2, 8, 4]
```

```
3×3 Matrix{Int64}:
```

```
2  5  3
```

```
5  8  6
```

```
3  6  4
```


Summary

1. Data frames
2. Random number generators
3. Statistics
4. Functional programming

Bonus: RCall

- ▶ Julia has a package called **RCall** that provides easy access to R, just press “\$” at the REPL.
- ▶ Or you can use macros to pass objects to R, and get them from R, and run calculations in R.
- ▶ Likewise, R has a package called **JuliaCall** to embed Julia in R.
- ▶ Similar bridges operate to python. It's good to talk.