Scientific Programming in Julia

Performant and Elegant

Nicholas Gale & Stephen Eglen

Course Objectives

In this course we will cover:

- Why we want to learn Julia.
- How to interact with Julia and the basics.
- The key differences between Julia and R.
- Several advanced concepts: types and structures, optimisation, high performance computing.
- Some essential packages: Plotting, Statistics, Distributed, CUDA.

A new language?

- A new language needs to offer benefits over the existing language.
- Current paradigm is a two language development cycle - the *two-language problem*.
- This is not elegant and lots of code is rewritten.

Why Julia?

- Julia offers a solution to the two language problem: it is fast *and* flexible.
- It uses a method called just-in-time (JIT) compiliation to generate compiled code just before it is needed.
- Easy to develop in.

Interacting with Julia

- Three primary ways to interact with Julia: Notebooks, terminal, and REPL.
- The notebook format is through the "IJulia" package and Jupyter notebook software. This is exactly like Python and R notebooks but with Julia code.
- The terminal can be used to execute scripts with the bash command `julia path/to/script`.
- REPL stands for Read-Evaluate-Print-Loop and is an interactive Julia session initiated by running `julia` in terminal.

REPL

- The REPL can execute basic commands or run scripts with the include function:

```
include("my_first_script.jl")
```

```
The output of my first script, is a string!
```

- Scripting in the REPL is costly *once*. Scripting from terminal is costly *everytime*.
- `println(arg)` is the method for printing (print works, but doesn't generate a new line).
- `display(arg)` creates a user-friendly readout in the ```

- Working in the REPL is very intuitive: functions generally have very mathematical names.
- Variable assignment is done through the `=` operator, equality is tested through the `==` operator, and indistinguishability through the `===` operator. This is different to R which uses the `<-` assignment operator and the `identical(a,b)` function for indistinguishability.

```
a = 1.0
b = 1
println(a == b)
println(a === b)
```

```
true
false
```

Operators

- Base operators are the same as in R (+, -, *, ^, etc).
- `div(a,b)` and `mod(a,b)` get division without remainder, and remainder.
- Unlike R, Julia supports infix operators: +=, -=, *=, /=.
- Infix operators modifiy a variable in-place and can be very useful e.g. loops:

```
a = 7
a += 1
a
```

```
8
```

Vectors

- A vector is created with square brackets and is column delimited as opposed to R's `c(tuple...)`:

```
a = [1, 2, 4.0, 1]
```

```
4-element Vector{Float64}:
 1.0
 2.0
 4.0
 1.0
```

- A vector can be indexed through integer, logical, and cartesian indexes using square brackets `[]`. Ranges are covered by the `:` operator
- Logical indexes must be the same size as the array, unlike in R.

```
println(a[2])
println(a[[true, false, true, false]])
println(a[CartesianIndex(1)])
println(a[2:4])
```

```
2.0

[1.0, 4.0]
1.0
[2.0, 4.0, 1.0]
```

- Julia doesn't support negative indexing like in R
- It does have a keyword `end` which references the end of a vector and can be combined with arithmetics to create consistent negative indexing

```
a = [1,2,3,4,5,6,7]
println(a[2:2:(end-1)])
```

```
[2, 4, 6]
```

Attention to detail

What is this thing called a range? Beware in R:

```
x = 1:1e6
format( object.size(x), units='MB')
```

will say the object is 3.8Mb.

whereas in Julia

```
x = 1:1e6
sizeof(x)
```

```
48
```

This is an example of an iterator.

Matrices

- A matrix is created with semi-colons to delimit rows and spaces (or `;;`) to delimit columns.

```
a =  [1 2 3;
    8 9 10;
    4 -30 0]
```

```
3×3 Matrix{Int64}:
 1    2   3
 8    9  10
 4  -30   0
```

- Matrices are column ordered so a linear index will access elements in the column first.

```
a = [1 2 3; 8 9 10; 4 -30 0];
println(a)
println(a[4])
println(a[CartesianIndex(2,1)])
println(a[2, 3])
println(a[1:2, 2:3])
```

```
[1 2 3; 8 9 10; 4 -30 0]
2
8
10
[2 3; 9 10]
```

Multi Dimensional Arrays

- Generally, `;` concatenates in the first dimension `;;` in the second and so on…
- Higher order arrays can be constructed like this, but it is generally cumbersome.
- Multidimensional indexes extend the rules for vectors and matrices: `[dim1, dim2, dim3...]`

```julia
a = [1; 2;; 3; 4;; 5; 6;;;
        7; 8;; 9; 10;; 11; 12]
display(a)
println(a[10])
println(a[CartesianIndex(2, 3, 2)])
```

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

[:, :, 2] =
 7   9  11
 8  10  12
10
12
```

General Arrays and Construction

- In Julia ranges can be created with steps like in R and are iterables but they are non-allocating (in R they create a vector).

```
a = 2:50:300
println(a[4])
println(a)
println(typeof(a))
```

```
152
2:50:252
StepRange{Int64, Int64}
```

Collect

- Vectors can be created with the `collect` function which works on any iterable collection.

```
a = collect(2:50:300)
println(a)
println(typeof(a))
```

```
[2, 52, 102, 152, 202, 252]
Vector{Int64}
```

Preallocation

- Preallocation can be done with `ones(dims...)` and `zeros(dims..)`

```
a = ones(2,3,5)
b = zeros(4,4)
display(a)
display(b)
```

```
2×3×5 Array{Float64, 3}:
[:, :, 1] =
 1.0  1.0  1.0
 1.0  1.0  1.0

[:, :, 2] =
 1.0  1.0  1.0
 1.0  1.0  1.0

[:, :, 3] =
 1.0  1.0  1.0
 1.0  1.0  1.0
```

```
[:, :, 4] =
 1.0  1.0  1.0
4×4 Matrix{Float64}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
```

Concatenation

- Arrays can be abstractally contactentated using space, and semilcolons `;`, `;;`, `;;;`.
- Horizontal concatentation is through `hcat(array1, array2, ...)`
- Vertical concatentation is through using `vcat(arrays...)`.

```
a = ones(3,3);
b = zeros(3,3);
display(hcat(a,b))
display(vcat(a,b))
```

```
3×6 Matrix{Float64}:
 1.0  1.0  1.0  0.0  0.0  0.0
 1.0  1.0  1.0  0.0  0.0  0.0
 1.0  1.0  1.0  0.0  0.0  0.0

6×3 Matrix{Float64}:
 1.0  1.0  1.0
 1.0  1.0  1.0
 1.0  1.0  1.0
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

Strings

- A character is indicated by apostrophes `'a'`.
- A string is a partial function from indexes to character literals. Unlike R apostrophes will not work, use double quotes `"a"` or triple double quotes `""" a """`.
- Triple double quotes allow some flexibility which can be useful in writing internal scripts:

```
"""The triple quotes allow for "quotes" to be embedded into a string."""
```

```
"The triple quotes allow for \"quotes\" to be embedded into a string."
```

String Indexing

- Strings can be indexed in Julia like in R and unlike R are not decoded into a character vector

```
a = "A very long string"
println(a[2:2:10])
```

```
eyln
```

Control Flow

- True and False are given by the `true` and `false` keywords.
- Logical true and false are: 0 and 1.
- Control flow blocks always have an `end` to terminate them. They are initiated with a special keyword. They don't usually have curly braces like in R.

```
println(true == false)
println(true == 1)
println(false == 0)
```

```
false
true
true
```

If, else, and elseif

- If blocks can have optional `elseif` and `else` keywords inside the block:

```julia
cond = true
if cond
    println("If this wasn't true, nothing would happen")
end
```

```
If this wasn't true, nothing would happen
```

```julia
condelse = 1
if condelse==2
    println("It's an even prime!")
else
    println("It wasn't true. The else block executed")
end
```

```
It wasn't true. The else block executed
```

```julia
condif = false
condelse = "three"
if condif == 1
    println("We miss this one")
elseif condelse == "three"
    println("The elseif block executes")
else
    println("The else block doesn't")
end
```

```
The elseif block executes
```

Ternary Operator

- The ternary operator can execute if one-liners: `conditional_expression ? iftrue_code_execute : else_code_execute`.

```
res = (8 < 4) ? "Maths broke." : "Situation Normal"
println(res)
```

```
Situation Normal
```

While

- While statements are also executed when a conditional expression evaluates as true. They can be useful in loops:

```julia
i = 0
while i < 10
    i+=1
    println("Not done yet..")
end
println("Done")
```

```
Not done yet..
Not done yet..
Not done yet..
Not done yet..
Not done yet..
Not done yet..
Not done yet..
Not done yet..
Not done yet..
Not done yet..
Done
```

For Loops

- For loops operate in the same way as R with the `for var in collection` structure.
- They can operate over any abstract collection or vector.
- The `in` operator can be replaced with `=`.

```
for i = 1:2
    println(i)
end
println()
for i in 20:22
    println(i)
end
```

```
1
2

20
```

List Comprehension

- Julia supports list comprehension: `[ expression(i) for i in collection]` generates a vector with objects specified by expression i.

```julia
vec = [i^2 for i in 1:4:33]
```

```
9-element Vector{Int64}:
    1
   25
   81
  169
  289
  441
  625
  841
 1089
```

Functions

- Functions are similarly defined as in R, but instead of setting the function name as an object of the function method we use a function block.
- Functions are defined with the keyword `function` followed by the specification of the function name and arguments.
- Functions have a return keyword which will return the moment the keyword is reached.

```
function complex_modulus1(x, iy)
    tmp = x^2 + iy^2
    return sqrt(tmp)
end
```

```
complex_modulus1 (generic function with 1 method)
```

- If there is no return keyword the function will return the last line.

- Functions can be written in one line in a mathematical format.

```
complex_modulus(x, iy) = sqrt(x^2 + iy^2)
```

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

- The recommended structure is to use a function block with a return keyword.

Splatting

- Variable arguments are given by the splat `...` operator.
- Can be used in function definition.

```
function g(x...)
    return x[end]
end
println(g(4,5,6))
```

```
6
```

- Splatting can also be used in calls to functions with multiple arguments.

```
function f(x1, x2, x3)
    return (x1*x2)-x3
end
fvec = [5,4,1.0];
println(f(fvec...))
```

```
19.0
```

- Unlike R `...` must be attached to a variable name

Anonymous Functions

- Anonymous functions in Julia are given using the `->` operator. This is *assignment* in R as opposed to the `function(x) exp_x` syntax in R.
- Functions are first class objects in Julia so these may be assigned variable names or exist as standalones

```
F = x -> x^2
println(F(2))
```

```
4
```