Introduction to Julia

JuliaAcademy

Instructor Info

Jane Herriman

Jane Herriman is Director of Diversity and Outreach at Julia Computing and a PhD student at Caltech.

She is a Julia, dance, and strength training enthusiast.

Course Info

Course duration: 85 min total

With review quizzes

Prereq: nil

Overview

In this course, we explore the fundamentals of Julia in an interactive manner via Jupyter Notebooks. Over the course of 13 video lectures, totalling 85 minutes, we cover Introduction, Basics: (Strings, Data structures, Review), Control Flow: (Loops, Conditionals, Functions), Packages and Plotting, and Advanced topics: (Multiple Dispatch, Julia is fast!, Basic Linear Algebra, Factorisations). Before we begin, here is a short introduction of The Julia Language:

- Julia is a general-purpose, high-performance language
- Julia looks like Python, feels like Lisp, and runs like C or Fortran
- Julia is easy-to-use, powerful, and performant
- Can be easily tested on https://juliabox.com

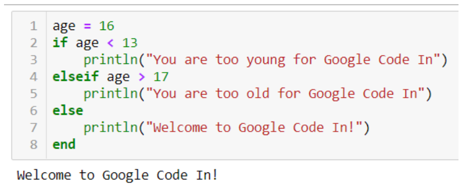
Course Summary

Basics

We focus on 4 data structures: strings, dictionaries, tuples and arrays. Interpolation and concatenation are 2 useful built-in functions on strings. `$` sign can be used to insert variables into a string for interpolation; `*` concatenates strings. On to the other data types, unlike tuples and arrays, dictionaries are unordered; unlike dictionaries and arrays, tuples are immutable (content cannot be altered after declaration). `push!` and `pop!` functions add and remove data in dictionaries and arrays. `typeof()` checks the data type of variables.

Control Flow

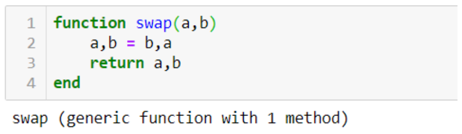
Julia variable scopes are divided into global and local, where those declared inside loops, conditionals and functions are local. Julia supports using $\in$ to replace $=$ when iterating in expressions like `for i = 1:10; ...; end`. Julia also supports multiple iterators in array comprehension syntax, where a nested for loop to populate matrix C of dimensions `(m,n)` can be unpacked into an elegant one-liner: `C = [i+j for i in 1:m, j in 1:n` for efficiency.



```
age = 16
if age < 13
    println("You are too young for Google Code In")
elseif age > 17
    println("You are too old for Google Code In")
else
    println("Welcome to Google Code In!")
end
```

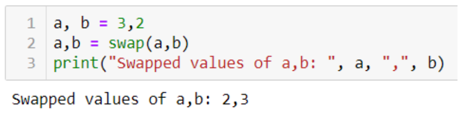
```
Welcome to Google Code In!
```

Julia conditionals follows the syntax in Figure 1. Another method of executing if-else logic in Julia is ternary operators. With the syntax `a ? b : c`, ternary operators takes in condition `a`, executes event `b` if `a` evaluates as `true`, and executes `c` if `a` evaluates as `false`. Logical operators `&&` (and), `||` (or), `!` (not) can simplify or short-circuit evaluations. An example is `a && println("This is true")`, where the message is only printed when `a == true`.



```
function swap(a,b)
    a,b = b,a
    return a,b
end
```

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



```
a, b = 3,2
a,b = swap(a,b)
print("Swapped values of a,b: ", a, ",", b)
```

```
Swapped values of a,b: 2,3
```

2 major function types in Julia are mutating & non-mutating functions. Mutating functions alter their inputs, while non-mutating ones do not. Broadcasting is also supported, and allows functions to be executed separately over each element of the input, and not over the whole input object. A mathematical equivalent of this is the Hadamard product (element-wise) versus normal matrix multiplication.

Introduction to Julia

JuliaAcademy

Instructor Info

Jane Herriman

Jane Herriman is Director of Diversity and Outreach at Julia Computing and a PhD student at Caltech.

She is a Julia, dance, and strength training enthusiast.

Course Info

Course duration: 85 min total

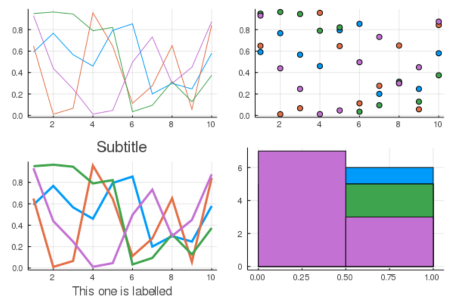
With review quizzes

Prereq: nil

Course Summary

Packages and Plotting

Julia packages extend the built-in functions. Packages are added by `using Pkg; Pkg.add("PACKAGE_NAME")`, and loaded by `using PACKAGE_NAME`. Here we focus on the Plots.jl package and plot some graphs. Experimenting with lines plots `plot(x,y)`, scatter plots `scatter!(x,y)`, subplots `plot(p1,p2,p3,p4, layout=(2,2)` and others, we can derive sophisticated models such as:



Advanced topics

Multiple dispatch is one of Julia's most potent features, and allows different methods to be dispatched for the same function, depending on the input arguments. The list of methods can be printed by `method(generic function)`, while the `@which` macro verifies which method was dispatched. To recap let's tackle a problem: After importing + from Base, how do we extend it to compute the union of 2 arrays: $A \cup B$? Considering the type of A, B, we know the answer to be `+(a:Array, b::Array) = union(a,b)`.

On top of being powerful, Julia also runs fast. Here, we use BenchmarkTools to benchmark Julia against C and Python and observe that the Julia built-in `sum` function executes faster than both Python Numpy and C built-in functions.

Besides, Julia supports basic linear algebra, e.g. multiplication, transposition, solving linear systems for square matrices. For a linear system $\boldsymbol{A}x = b$, there is no unique solution when it is overdetermined, and infinitely many solutions when it is underdetermined. We return the least squares solution for overdetermined systems s.t. $||\boldsymbol{A}x - b||$ is minimised, and the solution of minimal norm for underdetermined systems.

To perform LU and QR factorisation in Julia, let's go over eigendecomposition and singular value decomposition (SVD). `lufact(A)` perform LU factorisation on $\boldsymbol{A}$. Other functions include `det()` for the determinant, `qrfact()` for QR factorisation. Eigendecomposition can be computed via `eigenfact()`. SVD can be computed via `svdfact()`, returning an SVD object as output, for which the least squares solution can be computed with the left division operator.