# Julia Tutorial

Brown University, CSCI 1810  
YounHun Kim, Samier Merchant  
Fall 2015

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2 Running Julia</td>
<td>3</td>
</tr>
<tr>
<td>3 Setting up Julia</td>
<td>3</td>
</tr>
<tr>
<td>4 Variables</td>
<td>4</td>
</tr>
<tr>
<td>5 Numbers</td>
<td>4</td>
</tr>
<tr>
<td>5.1 Ints, Floats</td>
<td>4</td>
</tr>
<tr>
<td>5.2 Booleans</td>
<td>4</td>
</tr>
<tr>
<td>6 Strings</td>
<td>4</td>
</tr>
<tr>
<td>6.1 Length</td>
<td>5</td>
</tr>
<tr>
<td>6.2 Extracting a single character</td>
<td>5</td>
</tr>
<tr>
<td>6.3 Extracting a substring</td>
<td>5</td>
</tr>
<tr>
<td>6.4 Concatenating two strings</td>
<td>5</td>
</tr>
<tr>
<td>7 Types</td>
<td>6</td>
</tr>
<tr>
<td>7.1 Declaring &amp; Asserting Types</td>
<td>6</td>
</tr>
<tr>
<td>7.2 Composite Types</td>
<td>6</td>
</tr>
<tr>
<td>7.3 Subtypes</td>
<td>6</td>
</tr>
<tr>
<td>7.4 Parametric Types</td>
<td>6</td>
</tr>
<tr>
<td>7.5 The Nothing Type</td>
<td>6</td>
</tr>
<tr>
<td>8 Functions</td>
<td>6</td>
</tr>
<tr>
<td>9 Methods</td>
<td>7</td>
</tr>
<tr>
<td>9.1 Subtypes</td>
<td>8</td>
</tr>
<tr>
<td>9.2 Operators are functions too!</td>
<td>8</td>
</tr>
<tr>
<td>10 Logical Operators</td>
<td>8</td>
</tr>
<tr>
<td>11 Control Flow</td>
<td>9</td>
</tr>
<tr>
<td>11.1 If Statements</td>
<td>9</td>
</tr>
<tr>
<td>11.2 If-Else Statements</td>
<td>9</td>
</tr>
<tr>
<td>11.3 For loop</td>
<td>9</td>
</tr>
<tr>
<td>11.4 While loop</td>
<td>10</td>
</tr>
<tr>
<td>11.5 break, continue</td>
<td>10</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>12</td>
<td>Arrays</td>
</tr>
<tr>
<td>12.1</td>
<td>Construction</td>
</tr>
<tr>
<td>12.2</td>
<td>Adding to an Array</td>
</tr>
<tr>
<td>12.3</td>
<td>Removing from an Array</td>
</tr>
<tr>
<td>12.4</td>
<td>Accessing an element</td>
</tr>
<tr>
<td>12.5</td>
<td>Length of an array</td>
</tr>
<tr>
<td>12.6</td>
<td>Traversal</td>
</tr>
<tr>
<td>13</td>
<td>2-D Arrays</td>
</tr>
<tr>
<td>13.1</td>
<td>Construction</td>
</tr>
<tr>
<td>13.2</td>
<td>Accessing an element</td>
</tr>
<tr>
<td>13.3</td>
<td>Insertion</td>
</tr>
<tr>
<td>13.4</td>
<td>Size</td>
</tr>
<tr>
<td>13.5</td>
<td>Traversal</td>
</tr>
<tr>
<td>14</td>
<td>Dictionaries</td>
</tr>
<tr>
<td>14.1</td>
<td>Initialization</td>
</tr>
<tr>
<td>14.2</td>
<td>Traversal</td>
</tr>
<tr>
<td>14.3</td>
<td>Insertion</td>
</tr>
<tr>
<td>14.4</td>
<td>Accessing an element</td>
</tr>
<tr>
<td>14.5</td>
<td>Deletion</td>
</tr>
<tr>
<td>14.6</td>
<td>Extracting from an array</td>
</tr>
</tbody>
</table>
1 Introduction

This document serves as a concise reference for students learning how to use Julia v0.3. It emphasizes features/quirks of the language that CS181 students will most likely need to know when starting out. Basic how-tos are covered, such as creating variables, defining functions, initializing arrays/hashmaps. This does not serve as a replacement for the actual documentation (http://julia.readthedocs.org/en/latest/manual/), but rather exists to ease the transition from Python, Java or Matlab which Brown CS students are more familiar with.

2 Running Julia

To open a julia terminal, simply type julia. Here is what you would see:

```
$ julia
   (_)_(_)_(_)_   A fresh approach to technical computing
   | | | | | | | | Documentation: http://docs.julialang.org
   | | | | | | | | Type "help()" for help.
   | | | | | | | | Version 0.3.8 (2015-04-30 23:40 UTC)
   | | | | | | | | x86_64-linux-gnu
julia>
```

If, instead of opening a terminal, you wish to run julia code from a file, myscript.jl, type julia myscript.jl.

```
# This is a comment.
# contents of hello.jl
println("Hello World!");
```

Output of terminal:

```
$ julia hello.jl
Hello World!
```

3 Setting up Julia

On the CS department’s machines, julia is already set up and ready to use. The class package, CS181 is also installed and ready to go. For setting up Julia to use on your computer at home, you have two options.

1. Install the command-line version of Julia directly from the website. (http://julialang.org/downloads/)

2. Visit JuliaBox (http://juliabox.org), and log in using your brown.edu google account. It sets up a temporary virtual environment with Julia installed for you to use in your web browser. A few notes on regarding JuliaBox:
   - Files can be synced using your Google Drive folders. You can also create your own files and folders by navigating the "IJulia" tab.
   - Local settings and packages on JuliaBox are not stored between sessions. To make the terminal set up your work environment automatically every time you log in, one must edit the .bashrc file. If you need help with this, don’t be afraid to ask the TAs.

To get the class package, open the julia terminal and type

```
Pkg.clone("https://github.com/ykim27/CS181.jl.git")
```

When doing this for the first time, it may take a few minutes to run. Here is more or less what you should see when you do this immediately after setting up Julia:
4 Variables

To create a variable:

```julia
x = 10
```

To print a variable to console from within Julia:

```julia
julia> print(x)
10
```

5 Numbers

Julia is a typed language, and numbers are no exception. Types are covered in section 7, but we can still discuss some types that you’ll commonly encounter.

5.1 Ints, Floats

The type of 10 is typically Int64. The type Int is an alias for Int64. (Int32 also exists, but for the most part we won’t have to worry about this.)

```julia
julia> typeof(10)
Int64
julia> Int
Int64
```

In contrast, the type of 10.0 is "Float64". There is no built-in alias "Float".

5.2 Booleans

A useful number type is Bool, of which there are two values: true and false.

```julia
julia> x = true;
julia> typeof(x)
Bool
```

6 Strings

String literals are specified using double quotes.

```julia
julia> x = "Hello World!";
julia> typeof(x)
ASCIIString (constructor with 2 methods)
```
As indicated above, the type of a string literal is `ASCIIString`. There are other string types, such as `String`, `UTF8String` and `UTF16String`. All of these are subtypes of `String`. Methods are covered in section 9.

### 6.1 Length

```Julia
julia> length(x)
12
```

### 6.2 Extracting a single character

```Julia
julia> x[1]
'H'
```

Notice in the above example, that we used "1" to indicate the first character and not "0". Look at what happens when we do `x[0]`:

```Julia
julia> x[0]
ERROR: BoundsError()
```

Arrays in Julia are 1-indexed, in contrast to Java, Python or C which are 0-indexed.

### 6.3 Extracting a substring

Use the colon (:) notation to specify the indices. The substring up to and including the last index is returned. The keyword `end` is reserved for the last index.

```Julia
julia> x[3:6]
"llo 

julia> x[3:end]
"llo World!

julia> x[1:3]
"Hel"

julia> x[3:end-3]
"llo Wor"
```

### 6.4 Concatenating two strings

Use `*` to concatenate two strings.

```Julia
julia> "ABC" * "123"
"ABC123"
```

When concatenating lots of strings (say, in a loop that runs hundreds or thousands of times), this can be quite slow. For students of CS181, our class package provides a convenient interface to do this via the `StringBuilder` type.

```Julia
julia> using CS181;
julia> sb = StringBuilder();
    for i = 1:10
        append(sb,i);
        end
    println(get_string(sb));
12345678910
```
7 Types

7.1 Declaring & Asserting Types

Use the double colon (::) syntax to:

1. Assert the type of arguments passed into a function. See section 8.
2. Declare the type of a variable stored in a composite type. See section 7.2

7.2 Composite Types

Int, Float, String are all examples of built-in types. More complicated types (somewhat similar to classes in Java/Python and structs in C) are referred to as composite types.

```julia
type MyCompositeType
  a :: Int
  b :: Float64
end
```

To create an object of type MyCompositeType, use its constructor:

```julia
x = MyCompositeType("xyz", 1, 2.0)
```

7.3 Subtypes

The symbol <: indicates that the type to the left of it is a subtype of the abstract type on the right. (In Julia, one cannot subtype a concrete type. Only abstract types are allowed to have subtypes.)

```julia
type MySubType <:: MyAbstractType
  v :: Int
end
```

7.4 Parametric Types

Some types are parametrized, sort of like how generics work in Java.

```julia
// JAVA Syntax
List<A>

# JULIA Syntax
Array(Int64, 1)
```

7.5 The Nothing Type

You probably shouldn’t be using this in your code, but just in case you run across this while debugging...

Julia’s equivalent of a "null" value is the singleton Nothing type. It is a "singleton", as in only one value of its type exists.

```julia
julia> x = Nothing()
julia> y = Nothing()
julia> x == y
true
```

8 Functions

Define a function using either of the following syntaxes. The latter is useful for functions that can be described in a single line of code.
function add(x,y)
    x + y
end

add(x,y) = x + y

To specify the types of the arguments:

function add(x::Number, y::Number)
    x + y
end

Julia code generally runs faster if types of the inputs and values are specified. Functions return the last computed value of a line without an ending semicolon, or whatever is specified using the return keyword.

f(x,y) = x + y
function g(x,y)
    return x * y
    x + y
end

julia> f(2,3)
5
julia> g(2,3)
6

You would use return for more complicated functions, such as ones that use if statements. It is also possible to return multiple values encapsulated in a tuple:

function F(x,y)
    return x,y
end

julia> first, second = F(3,"foo")
julia> first
3
julia> second
"foo"

9 Methods

Sometimes, functions come with multiple methods. Julia figures out which one to use depending on how many variables are passed in, and what types they are. The behavior is similar to what would happen if you overloaded a class’s method in Java. Here is an example of a function foo with two different methods:

function foo(x,y::Int)
    "Method 1"
end

function foo(x::Float64,y::Int)
    "Method 2"
end

and what happens when you call it with different kinds of arguments:
9.1 Subtypes

When it comes to types and subtypes, Julia looks for the header that is as specific as possible to the set of inputs you give it. For instance:

```julia
function myfunc(x::Number)
    println("I am a number")
end

function myfunc(x::Int)
    println("I am an integer")
end
```

1 is an `Int`, which is a subtype of `Number`. 1.0 is a `Float64`, which is also a subtype of `Number`. "a" is an `ASCIIString`, which is not a subtype of number. Observe what happens when we call `foo` on these inputs:

```julia
julia> myfunc(1)
I am an integer

julia> myfunc(1.0)
I am a number

julia> myfunc("a")
ERROR: 'myfunc' has no method matching myfunc(::ASCIIString)
```

9.2 Operators are functions too!

Basic operators (+, −, *, \^) in Julia are functions. Technically speaking, if we wanted to implement an "addition" function for our own composite types, we could define a method for the + function instead, like so:

```julia
julia> type MyType
    a::Int
end

julia> function +(x::MyType, y::MyType)
    MyType(x.a + y.a)
end
+ (generic function with 118 methods)

julia> MyType(3)+MyType(5)
MyType(8)
```

As expected, when you try to add things that can’t be added, you will get an error like this:

```julia
julia> 3+MyType(5)
ERROR: '+' has no method matching +(::Int64, ::MyType)
```

10 Logical Operators

The below operators all output a Bool value, "true" or "false".

1. Greater than: >
2. Less than: <
3. Is equal to: ==
4. Or: ||
5. And: &
6. Negation: !

These are pretty standard as far as logical operators go.

```julia
julia> x = 3; # set x equal to 3
julia> x > 2
true
julia> x < 1
false
julia> x == 10
false
julia> x == 10 || x == 3
true
julia> x > 1 && x < 5
true
julia> !false
true
```

## 11 Control Flow

### 11.1 If Statements

The syntax is `if(...)`, followed by `end`.

```julia
if x == 3
    println("x is 3")
end
```

### 11.2 If-Else Statements

The syntax is `if(...)`, then `else`, then `end`.

```julia
if x < y
    println("x is less than y")
else
    println("x is greater than or equal to y")
end
```

It is also possible to do an if-elseif statement:

```julia
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

### 11.3 For loop

There are two ways to use for loops. The first way is reminiscent of MATLAB’s syntax:

```julia
julia> for i = 1:4
    print(i)
end
1234
```
The phrase `1:4` is shorthand for a `Range` object containing the values from 1 to 4. To go backwards from 4 to 1, we specify the incremental value:

```julia
julia> for i = 4:-1:1
    print(i)
end
4321
```

The second way to use a for loop is to use the keyword `in` to iterate over a container. In this example, we iterate over an array.

```julia
julia> for i in [1,2,3,4]
    print(i)
end
1234
```

### 11.4 While loop

The while loop has the syntax `while`, followed by an `end`.

```julia
julia> i = 1;
julia> while i <= 4
    print(i)
    i += 1
end
1234
```

### 11.5 break, continue

`break` and `continue` are keywords that interrupt the flow of a loop. They are identical to what one would find in other languages. `break` stops the loop at the point of execution and prevents subsequent iterations, while `continue` makes the loop move to the next iteration.

The following loop only prints odd numbers from 1 to 4:

```julia
julia> for i=1:4
    if i%2 == 0
        continue
    end
    print(i)
end
13
```

This loop stops when it encounters a 3:

```julia
julia> for i=1:4
    if i == 3
        break
    end
    print(i)
end
12
```

### 12 Arrays

**Note:** Arrays are 1-indexed! Not 0-indexed like Java or Python!

#### 12.1 Construction

To create an array, use the following methods:

- `zeros(type, dims)`
- `Array (type, dims)`
12.2 Adding to an Array
To add to the end of an array:

```plaintext
push!(myArray, 2)
```

12.3 Removing from an Array
To remove the last element of the array:

```plaintext
pop!(myArray)
```
To remove an element from the array, given a index:

```plaintext
deleteat!(myArray, idx)
```

12.4 Accessing an element
To access a certain element of an array at a particular index:

```plaintext
println(myArray[1])
```

12.5 Length of an array

```plaintext
length(myArray)
```

12.6 Traversal
To traverse forwards with an index:

```plaintext
for i = 1:endof(myArray)
    println(myArray[i])
end
```
To traverse backwards with an index (traverse in reverse order):

```plaintext
for i = endof(myArray):-1:1
    println(myArray[i])
end
```
To traverse forwards without an index:

```plaintext
for element in a
    println(element)
end
```
13 2-D Arrays

13.1 Construction

There are a few ways to construct a 2D array:

  This creates the 2-d array \[
  \begin{bmatrix}
  1 & 2 \\
  3 & 4 \\
  \end{bmatrix}
  \].

- `Array(Type, Dim1, Dim2)`
  This creates an uninitialized empty array of the specified size and type. **warning:** The initial values are unpredictable. If you wish to create an array initialized to all zeros, use the function `zeros` instead.

```
julia> Array(Int,2,3)
2x3 Array{Int64,2}:
   -4  -4  -4
  90935264  3577  91964099
julia> zeros(Int,2,3)
2x3 Array{Int64,2}:
   0   0   0
   0   0   0
```

13.2 Accessing an element

`myArray[x,y]`

13.3 Insertion

`myArray[x,y] = new_value`

13.4 Size

To get the number of rows and columns of your 2D array:

```
(rows, cols) = size(myArray)
```

To get only the number of rows:

```
rows = size(myArray,1)
```

To get only the number of columns:

```
cols = size(myArray,2)
```

13.5 Traversal

2D Arrays are stored column wise in memory in Julia. This means that the array \[
\begin{bmatrix}
  1 & 2 \\
  3 & 4 \\
\end{bmatrix}
\] is stored as a list of two columns: [1,3] and [2,4]. Note the traversal order:

```
julia> for element in myArray
    print(element)
end
1324
```

If you want control over the indices:
for x = 1:size(a,1)
   for y = 1:size(a,2)
       println(myArray[x,y])
   end
end

However, the below code is slightly faster, since we are traversing rows in the innermost loop:

for y = 1:size(a,2)
   for x = 1:size(a,1)
       println(myArray[x,y])
   end
end

14  Dictionaries

Dictionaries in Julia are functionally identical to dictionaries from python and hashmaps from Java.

14.1 Initialization

To create a dictionary:

- Dict()
- Dict{KeyType, ValueType}()
- Dict([(Key1, Value1), (Key2, Value2), …, (KeyN, ValueN)])

```julia
myDict = Dict()
myDict = Dict{{"A", 1}, {"B", 2}}
myDict = Dict{String, Int}()
```

14.2 Traversal

```julia
for (key, value) in myDict
    println(key)
    println(value)
end
```

14.3 Insertion

Syntax is the same as that of arrays.

```julia
mydict["C"] = 4;
```

14.4 Accessing an element

```julia
myDict["A"]
```

14.5 Deletion

```julia
delete!(collection, key).
delete!(mydict,"A")
```
14.6 Extracting from an array

- keys(collection) → Iterator
  
  ```julia
  for k in keys(mydict)
      print(k)
  end
  ```

- values(collection) → Iterator
  
  ```julia
  for v in values(mydict)
      print(v)
  end
  ```

- haskey(collection, key) → Bool, true if the key exists in the dict; false otherwise.
  
  ```julia
  julia> haskey(mydict,"Z")
  false
  ```

- get!(collection, key, default) → Element or default value if the element does exist. The difference is highlighted below. One throws an error, the other does not.
  
  ```julia
  julia> get(mydict,"Z",123)
  123
  julia> mydict["Z"]
  ERROR: key not found: "C"
  in getindex at dict.jl:617
  ```