FINC-672 – WORKSHOP IN FINANCE: EMPIRICAL RESEARCH

Julia Basics

Prof. Matt Fleckenstein
University of Delaware

mflecken@udel.edu

Goals

- ☐ Basic Julia Syntax
- ☐ Variables
- ☐ Boolean Operators and Numeric Comparisons
- ☐ Functions
- ☐ Control Flow and Loops

Julia Basics

- Today, we cover the basics of Julia as a programming language. Having a basic understanding of Julia will make you more *effective* and *efficient* in using Julia.
- This is going to be a very brief and *not* in-depth overview of the Julia language.
- If you are already familiar and comfortable with other programming languages, I encourage you to read Julia's documentation.[1]

[1] The docs are an excellent resource for taking a deep dive into Julia. It covers all the basics and corner cases, but it can be cumbersome. Especially, if you aren't familiar with software documentation.

Variables

- Variables are values that you tell the computer to store with an specific name, so that you can later recover or change its value.
- Julia has several types of variables but, the most important are[2]
  - Integers: `Int64`
  - Real Numbers: `Float64`
  - Boolean: `Bool`
  - Strings: `String`

[2] Integers and real numbers have by default 64 bits, that's why they have the `64` suffix in the name of the type. If you need more or less precision, there are `Int8` or `Int128` for example, where higher means more precision. Most of the time, this won't be an issue so you can just stick to the defaults.

Variables (Cont'd)

- We create new variables by writing the variable name on the left and its value in the right, and in the middle we use the = assignment operator.
- For example:

```
julia> name = "Julia"
"Julia"

julia> age  = 9
9
```

- Note that the return output of the last statement (`age`) was printed to the console.
- Here, we are defining two new variables: `name` and `age`.

Variables (Cont'd)

- If you want to define new values for an existing variable, you can repeat the steps in the assignment. Note that Julia will now override the previous variable's value with the new one. For example:

```
julia> age = 10
10
```

- We can also do operations on variables such as addition or division. For example, lets multiply `age` by 3:

```
julia> age * 3
30
```

Variables (Cont'd)

- We can also do operations on a variable and update it. For example,

```
julia> age = age + 7
17

julia> age
17
```

- We can inspect the types of variables by using the `typeof` function:

```
julia> typeof(age)
Int64
```

Boolean Operators and Numeric Comparisons

- Now that we've covered types, we can move to boolean operators and numeric comparison.
- We have three boolean operators in Julia:
  - `!`: **NOT**
  - `&&`: **AND**
  - `||`: **OR**

Boolean Operators and Numeric Comparisons (cont'd)

- Here are a few examples with some of them:

```
julia> !true
false
```

```
julia> (false && true) || (!false)
true
```

```
julia> (6 isa Int64) && (6 isa Real)
true
```

Boolean Operators and Numeric Comparisons (cont'd)

Regarding numeric comparison, Julia has three major types of comparisons:

1. Equality: either something is *equal* or *not equal* another
   - `==` "equal"
   - `!=` or `≠` "not equal"
2. Less than: either something is *less than* or *less than or equal to*
   - `<` "less than"
   - `<=` or `≤` "less than or equal to"
3. Greater than: either something is *greater than* or *greater than or equal to*
   - `>` "greater than"
   - `>=` or `≥` "greater than or equal to"

Boolean Operators and Numeric Comparisons (cont'd)

Here are some examples:

```
julia> 1 == 1
true
```

```
julia> 1 >= 10
false
```

Boolean Operators and Numeric Comparisons (cont'd)

- It evens works between different types:

```
julia> 1 == 1.0
true
```

- We can also mix and match boolean operators with numeric comparisons:

```
julia> (1 != 10) || (3.14 <= 2.71)
true
```

Functions

- Now that we already know how to define variables, let's turn our attention to **functions**.
- In Julia, a function **maps argument values to one or more return values**.

The basic syntax goes like this:

```
function function_name(arg1, arg2)
    result = stuff with the arg1 and arg2
return result
```

Functions (cont'd)

- The function declaration begins with the keyword `function` followed by the function name.
- Then, inside parentheses `()`, we define the arguments separated by a comma `,`.
- Inside the function, we specify what we want Julia to do with the parameters that we supplied.
  - All variables that we define inside a function are deleted after the function returns, this is nice because it is like an automatic cleanup.
  - After all the operations in the function body are finished, we instruct Julia to return the final result with the `return` statement.
- Finally, we let Julia know that the function definition is finished with the `end` keyword.

Creating New Functions

- Let's dive into some examples. First, let's create a new function that adds numbers together:

```
julia> function add_numbers(x, y)
    return x + y
end
add_numbers (generic function with 1 method)
```

Creating New Functions (cont'd)

- Now, we can use our `add_numbers` function:

```
julia> add_numbers(17, 29)
46
```

- And it works also with floats:

```
julia> add_numbers(3.14, 2.72)
5.86
```

Functions with Multiple Return Values

- A function can, also, return two or more values.
- See the new function `add_multiply` below:

```
julia> function add_multiply(x, y)
    addition = x + y
    multiplication = x * y
    return addition, multiplication
end
add_multiply (generic function with 1 method)
```

Functions with Multiple Return Values (cont'd)

In that case, we can do two things:

1. We can, analogously as the return values, define two variables to hold the function return values, one for each return value:

```
julia> return_1, return_2 = add_multiply(1, 2)
(3, 2)

julia> return_2
2
```

2. Or we can define just one variable to hold the function return values and access them with either `first` or `last`:

```
julia> all_returns = add_multiply(1, 2)
(3, 2)

julia> last(all_returns)
2
```

Conditional If-Else-Elseif

- In most programming languages, the user is allowed to control the computer's flow of execution.
- Depending on the situation, we want the computer to do one thing or another.
- In Julia we can control the flow of execution with `if`, `elseif` and `else` keywords. These are known as conditional statements.

Conditional If-Else-Elseif (cont'd)

- `if` keyword prompts Julia to evaluate an expression and depending whether `true` or `false` certain portions of code will be executed.
- We can compound several `if` conditions with the `elseif` keyword for complex control flow.
- Finally, we can define an alterative portion to be executed if anything inside the `if` or `elseif`'s is evaluated to `true`.
- This is the purpose of the `else` keyword.
- Finally, like all the previous keyword operators that we saw, we must tell Julia when the conditional statement is finished with the `end` keyword.

Conditional If-Else-Elseif (cont'd)

- Here's an example with all the **if-elseif-else** keywords:

```
julia> a = 1
1

julia> b = 2
2

julia>
if a < b
    "a is less than b"
elseif a > b
    "a is greater than b"
else
    "a is equal to b"
end
"a is less than b"
```

Conditional If-Else-Elseif (cont'd)

We can even wrap this in a function called `compare`:

```
julia> function compare(a, b)
    if a < b
        "a is less than b"
    elseif a > b
        "a is greater than b"
    else
        "a is equal to b"
    end
end
compare (generic function with 1 method)

julia>
compare(3.14, 3.14)
"a is equal to b"
```

For Loop

- The classical for loop in Julia follows a similar syntax as the conditional statements.
- You begin with a keyword, in this case `for`.
- Then, you specify what Julia should "loop" for, i.e., a sequence.
- Also, like everything else, you must finish with the `end` keyword.

For Loop (cont'd)

So, to make Julia print every number from 1 to 10, you can use the following for loop:

```
julia> for i in 1:10
    println(i)
end
1
2
3
4
5
6
7
8
9
10
```

While Loop

- The while loop is a mix of the previous conditional statements and for loops.
- Here, the loop is executed every time the condition is `true`.
- The syntax follows the same fashion as the previous one.
- We begin with the keyword **`while`**, followed by the statement to evaluated as either `true`.
- Like previously, you must end with the ‘end‘ keyword.

While Loop (cont'd)

Here's an example:

```
julia> n = 0
0

julia>
while n < 3
    global n += 1
end

julia>
n
3
```

While Loop (cont'd)

- As you can see, we have to use the `global` keyword.
- This is because of **variable scope**.
- Variables defined inside conditional statements, loops and functions exist only inside it.
- This is known as the **scope** of the variable.
- Here, we had to tell Julia that the `n` inside `while` loop is in the global scope with the `global` keyword.
- Finally, we used also the `+=` operator which is a nice short hand for `n = n + 1`.

Wrap-Up

- ☑ Basic Julia Syntax
- ☑ Variables
- ☑ Boolean Operators and Numeric Comparisons
- ☑ Functions
- ☑ Control Flow and Loops