Tutorial: Julia Basics

Contents

Overview

This tutorial will give some examples of basic Julia commands and syntax.

Getting Help

- Check out the official documentation for Julia: https://docs.julialang.org/en/v1/.
- Stack Overflow is a commonly-used resource for programming assistance.
- At a code prompt or in the REPL, you can always type `?functionname` to get help.

Comments

Comments hide statements from the interpreter or compiler. It's a good idea to liberally comment your code so readers (including yourself!) know why your code is structured and written the way it is. Single-line comments in Julia are preceded with a #. Multi-line comments are preceded with #= and ended with =#

Suppressing Output

You can suppress output using a semi-colon (;).

```
4+8;
```

That didn't show anything, as opposed to:

```
4+8
```

```
12
```

Variables

Variables are names which correspond to some type of object. These names are bound to objects (and hence their values) using the = operator.

```
x = 5
```

```
5
```

Variables can be manipulated with standard arithmetic operators.

```
4 + x
```

```
9
```

Another advantage of Julia is the ability to use Greek letters (or other Unicode characters) as variable names. For example, type a backslash followed by the name of the Greek letter (*i.e.* `\alpha`) followed by TAB.

```
α = 3
```

```
3
```

You can also include subscripts or superscripts in variable names using `\_` and `\^`, respectively, followed by TAB. If using a Greek letter followed by a sub- or super-script, make sure you TAB following the name of the letter before the sub- or super-script. Effectively, TAB after you finish typing the name of each `\character`.

```
β₁ = 10 # The name of this variable was entered with \beta + TAB + \_1 + TAB
```

```
10
```

However, try not to overwrite predefined names! For example, you might not want to use π as a variable name...

```
π
```

```
π = 3.1415926535897...
```

In the grand scheme of things, overwriting π is not a huge deal unless you want to do some trigonometry. However, there are more important predefined functions and variables that you may want to be aware of. Always check that a variable or function name is not predefined!

Data Types

Each datum (importantly, *not* the variable which is bound to it) has a data type. Julia types are similar to C types, in that they require not only the *type* of data (Int, Float, String, etc), but also the precision (which is related to the amount of memory allocated to the variable). Issues with precision won't be a big deal in this class, though they matter when you're concerned about performance vs. decimal accuracy of code.

You can identify the type of a variable or expression with the `typeof()` function.

```
typeof("This is a string.")
```

```
String
```

```
typeof(x)
```

```
Int64
```

Numeric types

A key distinction is between an integer type (or *Int*) and a floating-point number type (or *float*). Integers only hold whole numbers, while floating-point numbers correspond to numbers with fractional (or decimal) parts. For example, `9` is an integer, while `9.25` is a floating point number. The difference between the two has to do with the way the number is stored in memory. `9`, an integer, is handled differently in memory than `9.0`, which is a floating-point number, even though they're mathematically the same value.

```
typeof(9)
```

```
Int64
```

```
typeof(9.25)
```

```
Float64
```

Sometimes certain function specifications will require you to use a Float variable instead of an Int. One way to force an Int variable to be a Float is to add a decimal point at the end of the integer.

```
typeof(9.)
```

```
Float64
```

Strings

Strings hold characters, rather than numeric values. Even if a string contains what seems like a number, it is actually stored as the character representation of the digits. As a result, you cannot use arithmetic operators (for example) on this datum.

```
"5" + 5
```

```
MethodError: MethodError(+, ("5", 5), 0x0000000000006860)
MethodError: no method matching +(::String, ::Int64)
The function `+` exists, but no method is defined for this combination of argument types.

 [0mClosest candidates are:
 [0m  +(::Any, ::Any, [91m::Any [39m, [91m::Any... [39m)
 [0m [90m   @ [39m  [90mBase [39m  [90m [4moperators.jl:596 [24m [39m
 [0m  +( [91m::Missing [39m, ::Number)
 [0m [90m   @ [39m  [90mBase [39m  [90m [4mmissing.jl:123 [24m [39m
 [0m  +( [91m::Complex{Bool} [39m, ::Real)
 [0m [90m   @ [39m  [90mBase [39m  [90m [4mcomplex.jl:323 [24m [39m
 [0m  ...

Stacktrace:
  [1] top-level scope
 [90m   @ [39m  [90m~/Teaching/BEE4850/spring2026/tutorials/ [39m [90m [4mjulia-
basics.qmd:117 [24m [39m
```

However, you can try to tell Julia to interpret a string encoding a numeric character as a numeric value using the `parse()` function. This can also be used to encode a numeric data as a string.

```
parse(Int64, "5") + 5
```

```
10
```

Two strings can be concatenated using `*`:

```
"Hello" * " " * "there"
```

```
"Hello there"
```

Booleans

Boolean variables (or *Bools*) are logical variables, that can have `true` or `false` as values.

```
b = true
```

```
true
```

Numerical comparisons, such as ==, !=, or <, return a Bool.

```
c = 9 > 11
```

```
false
```

Bools are important for logical flows, such as if-then-else blocks or certain types of loops.

Mathematical operations

Addition, subtraction, multiplication, and division work as you would expect. Just pay attention to types! The type of the output is influenced by the type of the inputs: adding or multiplying an Int by a Float will always result in a Float, even if the Float is mathematically an integer. Division is a little special: dividing an Int by another Int will still return a float, because Julia doesn't know ahead of time if the denominator is a factor of the numerator.

```
3 + 5
```

```
8
```

```
3 * 2
```

```
6
```

```
3 * 2.
```

```
6.0
```

```
6 - 2
```

```
4
```

```
9 / 3
```

```
3.0
```

Raising a base to an exponent uses ^, not **.

```
3^2
```

```
9
```

Julia allows the use of updating operators to simplify updating a variable in place (in other words, using `x += 5` instead of `x = x + 5`.

Boolean algebra

Logical operations can be used on variables of type `Bool`. Typical operators are `&&` (and), `||` (or), and `!` (not).

```
true && true
```

```
true
```

```
true && false
```

```
false
```

```
true || false
```

```
true
```

```
!true
```

```
false
```

Comparisons can be chained together.

```
3 < 4 || 8 == 12
```

```
true
```

We didn't do this above, since Julia doesn't require it, but it's easier to understand these types of compound expressions if you use parentheses to signal the order of operations. This helps with debugging!

```
(3 < 4) || (8 == 12)
```

```
true
```

Data Structures

Data structures are containers which hold multiple values in a convenient fashion. Julia has several built-in data structures, and there are many extensions provided in additional packages.

Tuples

Tuples are collections of values. Julia will pay attention to the types of these values, but they can be mixed. Tuples are also *immutable*: their values cannot be changed once they are defined.

Tuples can be defined by just separating values with commas.

```
test_tuple = 4, 5, 6
```

```
(4, 5, 6)
```

To access a value, use square brackets and the desired index. **Note**: Julia indexing starts at 1, not 0!

```
test_tuple[1]
```

```
4
```

As mentioned above, tuples are immutable. What happens if we try to change the value of the first element of `test_tuple`?

```
test_tuple[1] = 5
```

```
MethodError: MethodError(setindex!, ((4, 5, 6), 5, 1), 0x0000000000006860)
MethodError: no method matching setindex!(::Tuple{Int64, Int64, Int64}, ::Int64, ::Int64)
The function `setindex!` exists, but no method is defined for this combination of argument
types.
Stacktrace:
 [1] top-level scope
[90m   @ [39m [90m~/Teaching/BEE4850/spring2026/tutorials/ [39m [90m [4mjulia-
basics.qmd:235 [24m [39m
```

Tuples also do not have to hold the same types of values.

```
test_tuple_2 = 4, 5., 'h'
typeof(test_tuple_2)
```

```
Tuple{Int64, Float64, Char}
```

Tuples can also be defined by enclosing the values in parentheses.

```
{juliA}
test_tuple_3 = (4, 5., 'h')
typeof(test_tuple_3)
```

Arrays

Arrays also hold multiple values, which can be accessed based on their index position. Arrays are commonly defined using square brackets.

```
test_array = [1, 4, 7, 8]
test_array[2]
```

```
4
```

Unlike tuples, arrays are mutable, and their contained values can be changed later.

```
test_array[1] = 6
test_array
```

```
4-element Vector{Int64}:
 6
 4
 7
 8
```

Arrays also can hold multiple types. Unlike tuples, this causes the array to no longer care about types at all.

```
test_array_2 = [6, 5., 'h']
typeof(test_array_2)
```

```
Vector{Any} (alias for Array{Any, 1})
```

Compare this with `test_array`:

```
typeof(test_array)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

Dictionaries

Instead of using integer indices based on position, dictionaries are indexed by keys. They are specified by passing key-value pairs to the `Dict()` method.

```
test_dict = Dict("A"=>1, "B"=>2)
test_dict["B"]
```

```
2
```

Comprehensions

Creating a data structure with more than a handful of elements can be tedious to do by hand. If your desired array follows a certain pattern, you can create structures using a *comprehension*. Comprehensions iterate over some other data structure (such as an array) implicitly and populate the new data structure based on the specified instructions.

```
[i^2 for i in 0:1:5]
```

```
6-element Vector{Int64}:
  0
  1
  4
  9
 16
 25
```

For dictionaries, make sure that you also specify the keys.

```
Dict(string(i) => i^2 for i in 0:1:5)
```

```
Dict{String, Int64} with 6 entries:
  "4" => 16
  "1" => 1
  "5" => 25
  "0" => 0
  "2" => 4
  "3" => 9
```

Functions

A function is an object which accepts a tuple of arguments and maps them to a return value. In Julia, functions are defined using the following syntax.

```
function my_actual_function(x, y)
    return x + y
end
my_actual_function(3, 5)
```

```
8
```

Functions in Julia do not require explicit use of a `return` statement. They will return the last expression evaluated in their definition. However, it's good style to explicitly `return` function outputs. This improves readability and debugging, especially when functions can return multiple expressions based on logical control flows (if-then-else blocks).

Functions in Julia are objects, and can be treated like other objects. They can be assigned to new variables or passed as arguments to other functions.

```
g = my_actual_function
g(3, 5)
```

```
8
```

```
function function_of_functions(f, x, y)
    return f(x, y)
end
function_of_functions(g, 3, 5)
```

```
8
```

Short and Anonymous Functions

In addition to the long form of the function definition shown above, simple functions can be specified in more compact forms when helpful.

This is the short form:

```
h₁(x) = x^2 # make the subscript using \_1 + <TAB>
h₁(4)
```

```
16
```

This is the anonymous form:

```
x->sin(x)
(x->sin(x))(π/4)
```

```
0.7071067811865475
```

Mutating Functions

The convention in Julia is that functions should not modify (or *mutate*) their input data. The reason for this is to ensure that the data is preserved. Mutating functions are mainly appropriate for applications where performance needs to be optimized, and making a copy of the input data would be too memory-intensive.

If you do write a mutating function in Julia, the convention is to add a `!` to its name, like `my_mutating_function!(x)`.

Optional arguments

There are two extremes with regard to function parameters which do not always need to be changed. The first is to hard-code them into the function body, which has a clear downside: when you do want to change them, the function needs to be edited directly. The other extreme is to treat them as regular arguments, passing them every time the function is called. This has the downside of potentially creating bloated function calls, particularly when there is a standard default value that makes sense for most function evaluations.

Most modern languages, including Julia, allow an alternate solution, which is to make these arguments *optional*. This involves setting a default value, which is used unless the argument is explicitly defined in a function call.

```
function setting_optional_arguments(x, y, c=0.5)
    return c * (x + y)
end
```

```
setting_optional_arguments (generic function with 2 methods)
```

If we want to stick with the fixed value $c = 0.5$, all we have to do is call `setting_optional_arguments` with the `x` and `y` arguments.

```
setting_optional_arguments(3, 5)
```

```
4.0
```

Otherwise, we can pass a new value for `c`.

```
setting_optional_arguments(3, 5, 2)
```

```
16
```

Passing data structures as arguments

Instead of passing variables individually, it may make sense to pass a data structure, such as an array or a tuple, and then unpacking within the function definition. This is straightforward in long form: access the appropriate elements using their index.

In short or anonymous form, there is a trick which allows the use of readable variables within the function definition.

```
h₂((x,y)) = x*y # enclose the input arguments in parentheses to tell Julia to expect and unpack a tuple
```

```
h₂ (generic function with 1 method)
```

```
h₂((2, 3)) # this works perfectly, as we passed in a tuple
```

```
6
```

```
h₂(2, 3) # this gives an error, as h₂ expects a single tuple, not two different numeric values
```

```
MethodError: MethodError(h₂, (2, 3), 0x000000000000686a)
MethodError: no method matching h₂(::Int64, ::Int64)
The function `h₂` exists, but no method is defined for this combination of argument types.

 [0mClosest candidates are:
 [0m  h₂(::Any)
 [0m [90m   @ [39m  [33mMain.Notebook [39m  [90m~/Teaching/BEE4850/spring2026/tutorials/
[39m [90m [4mjulia-basics.qmd:381 [24m [39m

Stacktrace:
 [1] top-level scope
 [90m   @ [39m  [90m~/Teaching/BEE4850/spring2026/tutorials/ [39m [90m [4mjulia-
basics.qmd:389 [24m [39m
```

```
h₂([3, 10]) # this also works with arrays instead of tuples
```

```
30
```

Vectorized operations

Julia uses **dot syntax** to vectorize an operation and apply it *element-wise* across an array.

For example, to calculate the square root of 3:

```
sqrt(3)
```

```
1.7320508075688772
```

To calculate the square roots of every integer between 1 and 5:

```
sqrt.([1, 2, 3, 4, 5])
```

```
5-element Vector{Float64}:
 1.0
 1.4142135623730951
 1.7320508075688772
 2.0
 2.23606797749979
```

The same dot syntax is used for arithmetic operations over arrays, since these operations are really functions.

```
[1, 2, 3, 4] .* 2
```

```
4-element Vector{Int64}:
 2
 4
 6
 8
```

Vectorization can be faster and is more concise to write and read than applying the same function to multiple variables or objects explicitly, so take advantage!

Returning multiple values

You can return multiple values by separating them with a comma. This implicitly causes the function to return a tuple of values.

```
function return_multiple_values(x, y)
    return x + y, x * y
end
return_multiple_values(3, 5)
```

```
(8, 15)
```

These values can be unpacked into multiple variables.

```
n, ν = return_multiple_values(3, 5)
n
```

```
8
```

```
ν
```

```
15
```

Returning `nothing`

Sometimes you don't want a function to return any values at all. For example, you might want a function that only prints a string to the console.

```
function print_some_string(x)
    println("x: $x")
    return nothing
end
print_some_string(42)
```

```
x: 42
```

Printing Text Output

The `Text()` function returns its argument as a plain text string. Notice how this is different from evaluating a string!

```
Text("I'm printing a string.")
```

```
I'm printing a string.
```

`Text()` is used in this tutorial as it *returns* the string passed to it. To print directly to the console, use `println()`.

```
println("I'm writing a string to the console.")
```

```
I'm writing a string to the console.
```

Printing Variables In a String

What if we want to include the value of a variable inside of a string? We do this using *string interpolation*, using `$variablename` inside of the string.

```
bar = 42
Text("Now I'm printing a variable: $bar")
```

```
Now I'm printing a variable: 42
```

Control Flows

One of the tricky things about learning a new programming language can be getting used to the specifics of control flow syntax. These types of flows include conditional if-then-else statements or loops.

Conditional Blocks

Conditional blocks allow different pieces of code to be evaluated depending on the value of a boolean expression or variable. For example, if we wanted to compute the absolute value of a number, rather than using `abs()`:

```
function our_abs(x)
    if x >= 0
        return x
    else
        return -x
    end
end
```

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

```
our_abs(4)
```

```
4
```

```
our_abs(-4)
```

```
4
```

To nest conditional statements, use `elseif`.

```
function test_sign(x)
    if x > 0
        return Text("x is positive.")
    elseif x < 0
        return Text("x is negative.")
    else
        return Text("x is zero.")
    end
end
```

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

```
test_sign(-5)
```

```
x is negative.
```

```
test_sign(0)
```

```
x is zero.
```

Loops

Loops allow expressions to be evaluated repeatedly until they are terminated. The two main types of loops are `while` loops and `for` loops.

While loops

`while` loops continue to evaluate an expression so long as a specified boolean condition is `true`. This is useful when you don't know how many iterations it will take for the desired goal to be reached.

```
function compute_factorial(x)
    factorial = 1
    while (x > 1)
        factorial *= x
        x -= 1
    end
    return factorial
end
compute_factorial(5)
```

```
120
```

While loops can easily turn into infinite loops if the condition is never meaningfully updated. Be careful, and look there if your programs are getting stuck. Also, If the expression in a `while` loop is false when the loop is reached, the loop will never be evaluated.

For loops

`for` loops run for a finite number of iterations, based on some defined index variable.

```
function add_some_numbers(x)
    total_sum = 0 # initialize at zero since we're adding
    for i=1:x # the counter i is updated every iteration
        total_sum += i
    end
    return total_sum
end
add_some_numbers(4)
```

```
10
```

`for` loops can also iterate over explicitly passed containers, rather than iterating over an incrementally-updated index sequence. Use the `in` keyword when defining the loop.

```
function add_passed_numbers(set)
    total_sum = 0
    for i in set # this is the syntax we use when we want i to correspond to different
container values
        total_sum += i
    end
    return total_sum
end
add_passed_numbers([1, 3, 5])
```

```
9
```

Linear algebra

Matrices are defined in Julia as 2d arrays. Unlike basic arrays, matrices need to contain the same data type so Julia knows what operations are allowed. When defining a matrix, use semicolons to separate rows. Row elements should not be separated by commas.

```
test_matrix = [1 2 3; 4 5 6]
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

You can also specify matrices using spaces and newlines.

```
test_matrix_2 = [1 2 3
                 4 5 6]
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

Finally, matrices can be created using comprehensions by separating the inputs by a comma.

```
[i*j for i in 1:1:5, j in 1:1:5]
```

```
5×5 Matrix{Int64}:
 1   2   3   4   5
 2   4   6   8  10
 3   6   9  12  15
 4   8  12  16  20
 5  10  15  20  25
```

Vectors are treated as 1d matrices.

```
test_row_vector = [1 2 3]
```

```
1×3 Matrix{Int64}:
 1  2  3
```

```
test_col_vector = [1; 2; 3]
```

```
3-element Vector{Int64}:
 1
 2
 3
```

Many linear algebra operations on vectors and matrices can be loaded using the `LinearAlgebra` package.

Package management

Sometimes you might need functionality that does not exist in base Julia. Julia handles packages using the `Pkg` package manager. After finding a package which has the functions that you need, you have two options: 1. Use the package management prompt in the Julia REPL (the standard Julia interface; what you get when you type `julia` in your terminal). Enter this by typing `]` at the standard green Julia prompt `julia>`. This will become a blue `pkg>`. You can then download and install new packages using `add packagename`. 2. From the standard prompt, enter `using Pkg; Pkg.add(packagename)`. The `packagename` package can then be used by adding `using packagename` to the start of the script.