Notes Set 1: Introduction

Chris Paciorek

2025-01-14

Table of contents

Introduction

This document is the first of a set of notes giving an overview of key syntax, tools, and concepts for using Julia. The notes are not meant to be particularly complete in terms of useful functions (Google and LLMs can now provide that quite well), but rather to introduce the language and consider key programming concepts in the context of Julia.

Given that, the document heavily relies on demos, with interpretation in some cases left to the reader.

This document covers basic syntax, basic types, data structures, and functions.

Variables and types

Basic types

Let's start by defining some variables and seeing what their types are.

```
typeof(2)
```

```
Int64
```

```
x = 2.0
```

```
2.0
```

```
typeof(x)
```

```
Float64
```

```
s = "hello"
```

```
"hello"
```

```
typeof(s)
```

```
String
```

```
typeof(s[1])
```

```
Char
```

```
typeof('\n')
```

```
Char
```

```
## Unicode characters
'h'
```

```
'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase)
```

```
'i'
```

```
'i': ASCII/Unicode U+0069 (category Ll: Letter, lowercase)
```

```
'\n'
```

```
'\n': ASCII/Unicode U+000A (category Cc: Other, control)
```

```
' '
```

```
' ': Unicode U+03B8 (category Ll: Letter, lowercase)
```

```
y = (3, 7.5)
```

```
(3, 7.5)
```

```
typeof(y)
```

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

As we'll be discussing more, knowing what type a variable is (particularly for large objects such as large arrays) is important for thinking about memory use, what methods work with what types of variables, and when variables need to be cast/coerced to a different type.

> **Warning**
>
> The Unicode/LaTeX characters may not show up in the PDF version of this document.

We can enter LaTeX characters/formatting by typing in LaTeX syntax (using a \) and then TAB.

```
 = 3.57  # \theta TAB
```

```
3.57
```

```
#=
Note the use of a comment
in the initial line.

And this here is a multi-line comment.
=#

x  = 7  # x\_1 TAB
```

```
7
```

A bit about strings

```
x = 'hello'

x = "hello"
x[1] = "a"
```

Casting/coercing between types

```
string(32)
```

```
"32"
```

```
parse(Float64, "32.5")
```

```
32.5
```

Some languages (such as R) will often cast between types behind the scenes. With Julia, one is often more deliberate about types as we'll see.

More on types and comparisons

```
x = 3
```

```
3
```

```
y = 3.0
```

```
3.0
```

```
x == y
```

```
true
```

```
x   y
```

```
false
```

```
x > y
```

```
false
```

```
x > y || x <= y
```

```
true
```

```
isa(x, Int)
```

```
true
```

```
y isa Int
```

```
false
```

```
y isa Number
```

```
true
```

```
'a'   "banana"  # \in TAB
```

```
true
```

```
'a'   "banana"  # \notin TAB
```

```
false
```

```
aString = "a"
```

```
"a"
```

```
'a' == aString
```

```
false
```

```
'a' == aString[1]
```

```
true
```

Conditional (if-else) statements

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

```
x and y are equal
```

Printing and string interpolation

We can use variables in print statements in various ways.

```
person = "Alice"
```

```
"Alice"
```

```
person = "Alice";

"Hello, $(person) with name of length $(length(person))."
```

```
"Hello, Alice with name of length 5."
```

```
println("Hello, ", person, " with name of length ", length(person), ".")
```

```
Hello, Alice with name of length 5.
```

```
println("Hello, $(person) with name of length $(length(person)).")
```

```
Hello, Alice with name of length 5.
```

```
println("Hello, " * person * " with name of length " * string(length(person)) * ".")
```

```
Hello, Alice with name of length 5.
```

Functions and operators

Operators (and not just for math/booleans)

```
value = 7;
value *= 3;
value
```

```
21
```

```
Value
```

```
ERROR: UndefVarError: `Value` not defined
```

```
x = 3
```

```
3
```

```
tmp = 7x   # Unlike any other language I know!
```

```
21
```

```
s * " there"
```

```
"hello there"
```

```
s^4
```

```
"hellohellohellohello"
```

Getting help on functions

Type `?` to get into help mode, then the name of the function you want help on.

To see all the functions/operators available in base Julia, type "Base." and hit tab.

Function definitions

```
function plus3(x=0)
  return 3+x
end
```

```
plus3 (generic function with 2 methods)
```

```
plus3(5)
```

```
8
```

Vectorized use

To use a function (or operator) in a vectorized way, we (with exceptions) need to use the dot notation.

```
y = [5.3, 2.5];
```

```
y + 3
plus3(y)
```

```
ERROR: MethodError: no method matching +(::Vector{Float64}, ::Int64)
For element-wise addition, use broadcasting with dot syntax: array .+ scalar
```

```
y .+ 3
```

```
2-element Vector{Float64}:
 8.3
 5.5
```

```
plus3.(y)
```

```
2-element Vector{Float64}:
 8.3
 5.5
```

```
## Apparently no general "recycling"/broadcasting.
x = [2.1, 3.1, 5.3, 7.9]
x .+ [0., 100.]
```

```
ERROR: DimensionMismatch: arrays could not be broadcast to a common size; got a dimension with length
```

Positional and keyword arguments

Positional arguments (which are matched based on the order they are given) are specified before keyword arguments.

```
function norm(x, p; verbose, extra)
  if verbose  # We'll see that "logging" is a better way to do this.
     println("Executing $(p)-norm.")
  end
  if !isfinite(p) and p > 0
    return maximum(abs.(x))
  return sum(x .^ p)^(1/p)
end

z = [3.3, 4.7, -2.2]

norm(z, 2, verbose=false, extra=0)
norm(z, 2; verbose=false, extra=0)
norm(z, 2, false, 0)
norm(z, p=1; verbose=false, extra=0)
norm(z, 1, extra=0, verbose=false)
```

Arguments can have defaults:

```julia
function norm(x, p=2; verbose=false)
  if verbose  # We'll see that "logging" is a better way to do this.
     println("Executing $(p)-norm.")
  end
  return sum(x .^ p)^(1/p)
end
```

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

> **Exercise**
>
> Try out various argument orders and giving or not giving names or values to the arguments and try to figure out the syntax rules of how Julia behaves. Think about how they are similar/different to your primary language and whether you like the syntax rules.

Keyword arguments are generally used for controlling function behavior rather than as core inputs. They are not involved in multiple dispatch (more later).

Try asking a ChatBot to write a norm function in Julia. When I tried Claude it gave me a better answer than what I wrote above (handling more cases for values of p and with a docstring.

> **Exercise**
>
> Write a function that implements the gamma density,
>
> $$f(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} \exp(-\beta x),$$
>
> for shape α and rate β or scale $1/\beta$, with $x > 0, \alpha > 0, \beta > 0$. Allow it to handle either the rate or scale parameterization and to return either the density or log density. Check that it works in a vectorized way for the random variable value and the parameters. Compare what you wrote to what a ChatBot gives.

Shorthand function definitions

These can be handy, but as a newcomer to Julia, I find them a bit hard to read.

```julia
plus3a(x=1) = 3+x

plus3b = (x=1) -> 3+x

# An anonymous function (useful for maps, functional programming).
((x) -> 3+x)

((x) -> 3+x)(7)
```

Arrays and subsetting

Sequences (and slicing)

```
some_text = "This is the Greek  "
```

```
"This is the Greek  "
```

```
some_text[1]
```

```
'T': ASCII/Unicode U+0054 (category Lu: Letter, uppercase)
```

```
some_text[19]
```

```
' ': Unicode U+03B8 (category Ll: Letter, lowercase)
```

```
some_text[1:4]
```

```
"This"
```

```
some_text[17:end]
```

```
"k  "
```

```
y = [1.1, 2.1, 3.2, 4.3, 5.7]
```

```
5-element Vector{Float64}:
 1.1
 2.1
 3.2
 4.3
 5.7
```

```
println(y)              # Original vector
```

```
[1.1, 2.1, 3.2, 4.3, 5.7]
```

```
println(y[1:3])         # First 3 elements
```

```
[1.1, 2.1, 3.2]
```

```
println(y[1:2:4])       # All odd-numbered elements
```

```
[1.1, 3.2]
```

```
println(y[end:-1:2])    # From end back to second element in reverse
```

```
[5.7, 4.3, 3.2, 2.1]
```

```
println(y[4:3])         # Empty subset
```

```
Float64[]
```

```
z = y[:]                # All elements (copy (not alias) of original vector)
```

```
5-element Vector{Float64}:
 1.1
 2.1
 3.2
 4.3
 5.7
```

```
println(y[[4,2,4,3,3]])  # Slice by index
```

```
[4.3, 2.1, 4.3, 3.2, 3.2]
```

```
y[[true,false,true,false,true]]   # Slice by boolean array
```

```
3-element Vector{Float64}:
 1.1
 3.2
 5.7
```

> **Exercise**
>
> Experiment more with slicing/indexing to make sure you get it, and what errors can occur. (As an example what happens if you index beyond the extent of the object?

Note that the discussion of `fruits[len]` in Section 7 of Think Julia is incorrect.

Arrays (i.e., lists)

```
x = ["spam", 2.0, 5, Missing, [10, 20], NaN]
```

```
6-element Vector{Any}:
    "spam"
   2.0
   5
    Missing
    [10, 20]
 NaN
```

```
length(x)
```

```
6
```

```
typeof(x)
```

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

```
y = [10, 20, 30, 40]
```

```
4-element Vector{Int64}:
 10
```

```
 20
 30
 40
```

```julia
typeof(y)
```

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

```julia
x[1] = 3.3
```

```
3.3
```

```julia
x[4] = 2.7
```

```
2.7
```

```julia
typeof(x)   # Mutable, but type doesn't change.
```

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

> **Math with arrays**
>
> For computational efficiency, we'd want the array to contain elements all of the same type. Note that languages like R and Python distinguish types intended for math (e.g., numpy arrays, R matrices) from more general types (e.g., lists). This is not the case for Julia, where the key thing is the type(s) involved.

Multi-dimensional arrays

```julia
A = [1 2 3; 4 5 6; 7 8 9]
```

```
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
```

```julia
A
```

```
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
```

```julia
A[2,2]
```

```
5
```

```julia
A[2,:]
```

```
3-element Vector{Int64}:
 4
```

```
 5
 6
```

```julia
size(A)
```

```
(3, 3)
```

```julia
size(A, 2)
```

```
3
```

```julia
## Defined column-wise:
A = [1:4  5:8  ones(Int64,4)]
```

```
4×3 Matrix{Int64}:
 1  5  1
 2  6  1
 3  7  1
 4  8  1
```

Arrays vs. vectors

```julia
ones(5)
```

```
5-element Vector{Float64}:
 1.0
 1.0
 1.0
 1.0
 1.0
```

```julia
ones(5, 1)
```

```
5×1 Matrix{Float64}:
 1.0
 1.0
 1.0
 1.0
 1.0
```

```julia
ones(1, 5)
```

```
1×5 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0
```

```julia
ones(5, 5)
```

```
5×5 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
```

```
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
```

```
## Outer product:
ones(5, 1) * ones(1, 5)
```

```
5×5 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
```

```
ones(5, 1) .* ones(1, 5)
```

```
5×5 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
```

A bit of linear algebra

We do linear algebra directly on the core Array type.

```
A = [1 2 3; 4 1 6; 7 8 1]
```

```
3×3 Matrix{Int64}:
 1  2  3
 4  1  6
 7  8  1
```

```
A * A
```

```
3×3 Matrix{Int64}:
 30  28  18
 50  57  24
 46  30  70
```

Much more in a few weeks.

More on vectorization

```
x = ["spam", 2.0, 5, [10, 20]]
```

```
4-element Vector{Any}:
  "spam"
 2.0
```

```
 5
  [10, 20]
```

```
length(x)
```

```
4
```

```
length.(x)
```

```
4-element Vector{Int64}:
 4
 1
 1
 2
```

```
map(length, x)
```

```
4-element Vector{Int64}:
 4
 1
 1
 2
```

```
x = [2.1, 3.1, 5.3, 7.9]
```

```
4-element Vector{Float64}:
 2.1
 3.1
 5.3
 7.9
```

```
x .+ 10
```

```
4-element Vector{Float64}:
 12.1
 13.1
 15.3
 17.9
```

```
x + x
```

```
4-element Vector{Float64}:
  4.2
  6.2
 10.6
 15.8
```

```
x .> 5.0
```

```
4-element BitVector:
 0
```

```
 0
 1
 1
```

```julia
x .== 3.1
```

```
4-element BitVector:
 0
 1
 0
 0
```

Reduction

```julia
A = rand(4, 5)
```

```
4×5 Matrix{Float64}:
 0.404951  0.523761  0.152349  0.868369   0.725839
 0.899675  0.855515  0.696137  0.0700364  0.564429
 0.658895  0.766265  0.394122  0.781437   0.966373
 0.801055  0.144414  0.18882   0.686381   0.320927
```

```julia
sum(A)
```

```
11.469749008141672
```

```julia
sum(A, dims = 1)   # 2D array result
```

```
1×5 Matrix{Float64}:
 2.76458  2.28996  1.43143  2.40622  2.57757
```

```julia
sum(A, dims = 1)[:]   # 1D array result
```

```
5-element Vector{Float64}:
 2.764576033786707
 2.289955287524771
 1.4314270033850072
 2.4062226391882433
 2.5775680442569437
```

```julia
sum(A, dims = 2)
```

```
4×1 Matrix{Float64}:
 2.6752688595347127
 3.0857913535168104
 3.567092575726648
 2.1415962193635014
```

List comprehension (comprehension syntax)

Similar to Python.

```
ysq = [ w^2 for w in y ]
```

```
4-element Vector{Int64}:
  100
  400
  900
 1600
```

```
xsqu = [ x^2 for x = 1:5 ]
```

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

```
xsqu_even = [ x^2 for x = 1:5 if iseven(x)]
```

```
2-element Vector{Int64}:
  4
 16
```

```
norm2 = [ x^2 + y^2 for x = 1:5, y = 1:5 ]
```

```
5×5 Matrix{Int64}:
  2   5  10  17  26
  5   8  13  20  29
 10  13  18  25  34
 17  20  25  32  41
 26  29  34  41  50
```

A nice terse shorthand but can be hard to read.

(Some people love it and some people hate it.)

Dictionaries, tuples, and structs

Dictionaries

Key-value pairs like Python dictionaries (and somewhat like named R lists).

```
x = Dict("test" => 3, "tmp" => [2.1, 3.5], 7 => "weird")
```

```
Dict{Any, Any} with 3 entries:
  7      => "weird"
  "test" => 3
```

```
  "tmp"  => [2.1, 3.5]
```

```
x["tmp"][2]
```

```
3.5
```

```
x[7]
```

```
"weird"
```

```
x["newkey"] = 'a'
```

```
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

```
keys(x)
```

```
KeySet for a Dict{Any, Any} with 4 entries. Keys:
  7
  "test"
  "tmp"
  "newkey"
```

```
x["hello"]
```

```
ERROR: KeyError: key "hello" not found
```

```
get(x, "hello", 0)
```

```
0
```

Note that the keys don't have to be strings! This could be good for caching/memoizing/lookup:

```
x = Dict(["foo", "bar"] => 3, "tmp" => [2.1, 3.5], 7 => "weird")
```

```
Dict{Any, Any} with 3 entries:
  7              => "weird"
  ["foo", "bar"] => 3
  "tmp"          => [2.1, 3.5]
```

```
x[["foo", "bar"]]
```

```
3
```

```
ind = 7
```

```
7
```

```
x[ind]
```

```
"weird"
```

What do you think will happen here?

```
ind = Int32(7)  # What do you expect?
x[ind]
```

```
ind = 7.0        # What do you expect?
x[ind]
```

Tuples

Tuples are are similar to 1-dimensional arrays but they are immutable (they can't be modified) and can have named elements.

```
x = (3, 5, "hello")
```

```
(3, 5, "hello")
```

```
x[2]
```

```
5
```

```
x[2] = 7
```

```
ERROR: MethodError: no method matching setindex!(::Tuple{Int64, Int64, String}, ::Int64, ::Int64)
```

```
x = 3
```

```
3
```

```
y = 9
```

```
9
```

```
y,x = x,y
```

```
(3, 9)
```

```
# Named tuple:
x = (a=3, b=5, other="hello")
```

```
(a = 3, b = 5, other = "hello")
```

```
x.b
```

```
5
```

What do you think will happen here?

```
x = (a=3, b=5, other="hello", b="foo")
x.b
```

Tuples come in handy for providing flexibility in function inputs and outputs, as seen next.

Tuples and functions

Here we create a function that can take an arbitrary number of inputs.

```
function flexsum(args...)
   println("The first value is $(args[1]).")
```

```
    return sum(args)
end
```

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

```
flexsum(5, 7, 9)
```

```
The first value is 5.
```

```
21
```

Here's how to call a function that takes multiple inputs, but pass as a tuple:

```
function mydiv(x, y)
    return x / y
end
```

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

```
vals = [3,5]
```

```
2-element Vector{Int64}:
 3
 5
```

```
mydiv(vals...)
```

```
0.6
```

We use tuples to have a function return multiple values.

```
function test()
  return 3, 5, [3,7]
end
```

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

```
test()
```

```
(3, 5, [3, 7])
```

Structs

A struct is a "composite type", a collection of named fields, useful for holding information of a particular structure.

```
struct Person
  name
  age
  occupation
end
```

```
lincoln = Person("Abraham Lincoln", 203, "politician")
```

```
Person("Abraham Lincoln", 203, "politician")
```

```
lincoln.age
```

```
203
```

We'll see much more on structs next week when we talk more about using types for robust code.

Loops

```
numThrows = 1000;
in_circle = 0;

# Run Monte Carlo simulation
for _ in 1:numThrows
  # Generate random points on 2x2 square.
  xPos = rand() * 2 - 1.0  # Equivalent to random.uniform(-1.0, 1.0)
  yPos = rand() * 2 - 1.0

  # Is point inside unit circle?
  if sqrt(xPos^2 + yPos^2) <= 1.0  # Equivalent to math.hypot()
    in_circle += 1
  end
end

# Estimate PI
pi_estimate = 4 * in_circle / numThrows
```

Variables defined in the loop are local variables accessible only in the scope of the loop (more on this soon). This avoids clutter in the global scope.

```
xPos
```

```
ERROR: UndefVarError: `xPos` not defined
```

We can iterate over elements of an object like this:

```
for i in eachindex(x)
  println(i)
end
```

```
a
b
other
```

String processing and regular expressions

```
x = "The cat in the hat."
```

```
"The cat in the hat."
```

```
replace(x, "at"=>"")
```

```
"The c in the h."
```

```
x = "We found 999 red balloons."
```

```
"We found 999 red balloons."
```

```
replace(x, r"[0-9]+"=>"some")  # Regular expression.
```

```
"We found some red balloons."
```

```
'a'   "banana"
```

```
true
```

```
x = "We found 99 red balloons."
```

```
"We found 99 red balloons."
```

```
m = match(r"[0-9]+ ([a-z]+)", x)
```

```
RegexMatch("99 red", 1="red")
```

```
m.match
```

```
"99 red"
```

```
m.captures
```

```
1-element Vector{Union{Nothing, SubString{String}}}:
 "red"
```

```
m.offset
```

```
10
```