

## Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann

### Notebook 02

---

#### Julia - first contact

General information

Resources

Open Source

How to install and run Julia

Standard number types

Integers

Floating point numbers

Rational numbers

Irrational numbers

Complex numbers

Vectors

Construction by explicit list of elements:

Other Vector constructors

Ranges

Vector dimensions

Subvectors

Views

Dot operations

Matrices

Tensors

Linear Algebra

Control structures

Functions

Functions and vectors

Macros

# Julia - first contact

---

## General information

---

## Resources

- [Homepage](#)
- [Documentation](#)
- [Cheat Sheet](#)
- [WikiBook](#)
- [7 Julia Gotchas and How to Handle Them](#)

Use the [Cheat Sheet](#) to see a compact and rather comprehensive list of basic things Julia.

This notebook tries to give a first introduction to Julia, featuring standard features which can be found in other languages as well. More Julia-specific things come later.

## Open Source

- Julia is an Open Source project started at MIT
- Julia itself is distributed under an [MIT license](#)
  - packages often have different licenses
- Development takes place on [github](#)
- As of October 2021, more than 1200 contributors to the code
- The Open Source paradigm corresponds well to the fundamental requirement that scientific research should be [transparent and reproducible](#)

## How to install and run Julia

- Installation:
  - [Download](#) from julialang.org (recommended by Julia creators)
  - Installation via system package manager (yast, apt-get, homebrew...)
- Workflows:
  - Pluto notebooks in the browser
  - From command line: Edit source code in any editor
  - Julia plugin of Visual Studio code editor
  - Jupyter notebooks in the browser

# Standard number types

---

- Julia is a strongly typed language, so any variable has a type.
- Standard number types allow fast execution because they are supported in the instruction set of the processors
- Default types are autodected from expression
- The `typeof` function allows to detect the type of a variable
- The `sizeof` funtion detects the size in bytes of a variable (1 byte = 8 bit)

## Integers

Integer variables are used to represent integer numbers.

```
i = 10
```

- `i=10`

```
Int64
```

- `typeof(i)`

```
8
```

- `sizeof(i)`

Besides of the default `Int64` type, Julia knows `Int8`, `Int16`, `Int32`

```
10
```

- `j::Int16=10`

```
Int16
```

- `typeof(j)`

```
2
```

- `sizeof(j)`

## Floating point numbers

Floating point variables are used to represent real numbers up to some precision.

```
x = 10.0
```

- `x=10.0`

```
Float64
```

- `typeof(x)`

8

- `sizeof(x)`

There is also `Float16`, `Float32`

144.3

- `y::Float32=144.3`

4

- `sizeof(y)`

## Rational numbers

 $r = 3//7$ 

- `r=3//7`

`Rational{Int64}`

- `typeof(r)`

16

- `sizeof(r)`

## Irrational numbers

 $\pi = 3.1415926535897\dots$ 

- `\pi`

0

- `sizeof(\pi)`

`Irrational{:\pi}`

- `typeof(\pi)`

 $e = 2.7182818284590\dots$ 

- `e`

## Complex numbers

 $z = 17.5 + 3.0im$ 

- `z=17.5+3im`

`ComplexF64` (alias for `Complex{Float64}`)

- `typeof(z)`

16

- `sizeof(z)`

## Vectors

---

- Elements of a given type stored contiguously in memory
- Vectors and 1-dimensional arrays are the same
- Vectors can be created for any element type
- Element type can be determined by `eltype` method

## Construction by explicit list of elements:

```
v1 = [1, 2, 3, 4, 5]
```

- `v1=[1,2,3,4,5]`

```
Int64
```

- `eltype(v1)`

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

- `typeof(v1)`

```
40
```

- `sizeof(v1)`

We can create a vector of floats:

```
v1f = [1.0, 2.0, 3.0, 4.0]
```

- `v1f=Float64[1,2,3,4]`

If one element in the initializer is float, the vector becomes float:

```
v2 = [1.0, 2.0, 3.0, 4.0]
```

- `v2=[1.0,2,3,4,]`

```
Vector{Float64} (alias for Array{Float64, 1})
```

- `typeof(v2)`

## Other Vector constructors

Create vectors of zeros, ones, constant, random or uninitialized values:

```
[0.0, 0.0, 0.0, 0.0, 0.0]
```

- `zeros(Float32,5)`

```
[1.0, 1.0, 1.0, 1.0, 1.0]
```

- `ones(5)`

```
[17.0, 17.0, 17.0, 17.0, 17.0, 17.0, 17.0, 17.0, 17.0, 17.0]
```

- `fill(17.0,10)`

```
Vector{Float16} (alias for Array{Float16, 1})
```

- `typeof(rand(Float16,20))`

```
[0.0, 0.0, -1.4423f15, 4.569f-41, -1.42506f15, 4.569f-41, 0.0, 0.0, 2.8f-44, 1.0f-45]
```

- `Vector{Float32}(undef,10)`

## Ranges

- Ranges describe sequences of numbers and can be used in loops, array constructors etc.
- They contain the recipe for the sequences, not the full data.

```
r1 = 1:10
```

- `r1=1:10`

```
UnitRange{Int64}
```

- `typeof(r1)`

Collect the sequence from a range into a vector:

```
w1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- `w1=collect(r1)`

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

- `typeof(w1)`

Add a step size to a range:

```
r2 = 1.0:0.8:9.8
```

- `r2=1:0.8:10`

```
StepRangeLen{Float64, Base.TwicePrecision{Float64}, Base.TwicePrecision{Float64}, Int
```

Create a vector from a list comprehension containing a range:

```
[0.841471, 0.891207, 0.932039, 0.963558, 0.98545, 0.997495, 0.999574, 0.991665, 0.973
```

```
• [sin(i) for i=1:0.1:5]
```

Create a random vector of given size:

Julia version of "linspace":

```
[0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0]
```

```
• collect(range(0,20,length=11))
```

## Vector dimensions

```
v6 = [1, 3, 5, 7, 9]
```

```
• v6=collect(1:2:10)
```

size is a tuple of dimensions

```
(5)
```

```
• size(v6)
```

length describes the overall length:

```
5
```

```
• length(v6)
```

## Subvectors

Copies of parts of vectors:

```
v7 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
• v7=collect(1:10)
```

```
subv7 = [2, 3, 4]
```

```
• subv7=v7[2:4]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
• subv7[1]=17;v7
```

## Views

```
v8 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- `v8=collect(1:10)`

```
subv8 = view(::Vector{Int64}, 2:4): [2, 3, 4]
```

- `subv8=view(v8,2:4)`

```
[1, 19, 3, 4, 5, 6, 7, 8, 9, 10]
```

- `subv8[1]=19;v8`

The `@views` macro can turn a copy statement into a view

```
v9 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- `v9=collect(1:10)`

```
view(::Vector{Int64}, 2:4): [2, 3, 4]
```

- `@views subv9=v9[2:4]`

```
[1, 29, 3, 4, 5, 6, 7, 8, 9, 10]
```

- `subv9[1]=29; v9`

## Dot operations

- Element-wise operations on vectors

```
v10 =
```

```
[0.0, 0.314159, 0.628319, 0.942478, 1.25664, 1.5708, 1.88496, 2.19911, 2.51327, 2.827
```

- `v10=collect(0:0.1π:2π)`

```
[0.0, 0.309017, 0.587785, 0.809017, 0.951057, 1.0, 0.951057, 0.809017, 0.587785, 0.30
```

- `sin.(v10)`

```
[100.0, 100.314, 100.628, 100.942, 101.257, 101.571, 101.885, 102.199, 102.513, 102.8
```

- `v10.+100`

## Matrices

- Elements of a given type stored contiguously in memory, with two-dimensional access
- Matrices and 2-dimensional arrays are the same



Zero initialization:

```
(5×6 Matrix{Float64}:           , 5×6 Matrix{Float64}:           , 5×6 Matrix{Int64}:
 0.0  0.0  0.0  0.0  0.0  0.0   1.0  1.0  1.0  1.0  1.0  1.0   17  17  17  17  17
 0.0  0.0  0.0  0.0  0.0  0.0   1.0  1.0  1.0  1.0  1.0  1.0   17  17  17  17  17
 0.0  0.0  0.0  0.0  0.0  0.0   1.0  1.0  1.0  1.0  1.0  1.0   17  17  17  17  17
 0.0  0.0  0.0  0.0  0.0  0.0   1.0  1.0  1.0  1.0  1.0  1.0   17  17  17  17  17
 0.0  0.0  0.0  0.0  0.0  0.0   1.0  1.0  1.0  1.0  1.0  1.0   17  17  17  17  17
```

- `zeros(5,6), ones(5,6), fill(17,5,6), rand(5,6)`

undef initialization:

```
3×3 Matrix{Float64}:
 5.0e-324  5.0e-324  0.0
 5.0e-324  0.0      0.0
 5.0e-324  0.0      0.0
```

- `Matrix{Float64}(undef,3,3)`

List comprehension:

```
m3 = 6×5 Matrix{Float64}:
 0.367879  0.606531  1.0      1.64872  2.71828
-0.153092 -0.252406 -0.416147 -0.68611 -1.1312
-0.240462 -0.396455 -0.653644 -1.07768 -1.77679
 0.353227  0.582373  0.96017  1.58305  2.61001
-0.0535265 -0.0882502 -0.1455  -0.239889 -0.39551
-0.308677 -0.508923 -0.839072 -1.3834  -2.28083
```

- `m3=[cos(x)*exp(y) for x=0:2:10, y=-1:0.5:1]`

The size of a matrix is the tuple of the two matrix dimensions:

```
(6, 5)
```

- `size(m3)`

The length of a matrix is the length of the contiguous storage array in memory:

```
30
```

- `length(m3)`

## Tensors

Tensors are arrays of dimension 3 and larger:

```

t3 = 3x4x3x4 Array{Float64, 4}:
[:, :, 1, 1] =
 0.155056  0.28608   0.840531  0.258298
 0.127177  0.626616  0.835662  0.135099
 0.976683  0.959982  0.234808  0.710854

[:, :, 2, 1] =
 0.521884  0.011436   0.845377  0.483208
 0.924438  0.874406   0.228565  0.297506
 0.275542  0.0683835  0.350426  0.674448

[:, :, 3, 1] =
 0.41195   0.0982991  0.482434  0.127802
 0.996988  0.361552  0.744331  0.15834
 0.732053  0.982214  0.786758  0.528707

[:, :, 1, 2] =
 0.609039  0.739848  0.60056   0.27588
 0.366695  0.880292  0.379202  0.358936
 0.274391  0.248146  0.118967  0.727726

[:, :, 2, 2] =
 0.200307  0.322285  0.936794  0.611028
 0.566803  0.566946  0.073731  0.778895
 0.433429  0.560232  0.0907034 0.0473299

[:, :, 3, 2] =
 0.620367  0.0702198  0.964224  0.16276
 0.757389  0.0649138  0.371296  0.660547
 0.766702  0.00408301 0.381515  0.174987

[:, :, 1, 3] =
 0.896548  0.94536   0.337234  0.172264
 0.227017  0.500550  0.105607  0.205017
 0.227017  0.500550  0.105607  0.205017

```

- `t3=rand(3,4,3,4)`

```
((3, 4, 3, 4), 144)
```

- `size(t3),length(t3)`

## Linear Algebra

- `using LinearAlgebra`

Let us create some linear algebra objects:

```
n = 10
```

- `n=10`

```
w =
```

```
[0.500465, 0.964304, 0.965805, 0.0358728, 0.06314, 0.636845, 0.197537, 0.226489, 0.32
```

- `w=rand(n)`

**u =**

[0.985701, 0.461082, 0.024454, 0.887917, 0.513778, 0.471111, 0.271964, 0.882261, 0.76

• **u=rand(n)****A =**

10×10 Matrix{Float64}:

0.37309	0.0391588	0.598725	0.756004	0.636856	...	0.804077	0.870187	0.0638292
0.192313	0.779179	0.391074	0.855619	0.942094		0.193337	0.540125	0.322208
0.197077	0.831407	0.99625	0.903965	0.102601		0.667509	0.421121	0.756866
0.204572	0.756074	0.302425	0.163882	0.914839		0.0120712	0.182038	0.801219
0.900581	0.87795	0.418617	0.108568	0.251656		0.907242	0.222466	0.520966
0.772484	0.533352	0.292527	0.672349	0.338092	...	0.597899	0.11918	0.603278
0.155588	0.453406	0.410177	0.923492	0.65403		0.118921	0.874301	0.758568
0.131551	0.967043	0.698563	0.655542	0.333738		0.961144	0.138891	0.17352
0.644305	0.291826	0.85641	0.204062	0.104068		0.585564	0.967923	0.0917366
0.20454	0.545154	0.539027	0.763523	0.198259		0.35373	0.513813	0.522379

• **A=rand(n,n)**Mean square norm  $\|u\|_2 = \sqrt{\sum_{i=1}^n u_i^2}$ :

2.1644114279972446

• **norm(u)**Dot product:  $(u, w) = \sum_{i=1}^n u_i w_i$ :

2.04080972762037

• **dot(u,w)**Matrix-vector product  $Au$ :

[3.15324, 3.00459, 3.56723, 2.08589, 3.42924, 3.16574, 3.32676, 2.62502, 2.9801, 2.78

• **A\*u**

Trace (sum of main diagonal elements):

5.245577462096071

• **tr(A)**

Determinant:

-0.008881272729455644

• **det(A)**

Inverse:

```
10×10 Matrix{Float64}:
```

```
 0.630388 -0.362444 -2.88854  1.64776  ... -0.831026  0.396407  7.6579
-2.81973  0.489814  0.743584 -1.66118  ...  2.41808  1.50169 -6.9686
 1.37109 -0.508897 -1.16438  2.57865  ... -0.972355  0.359166  7.29594
 0.644199  0.236785 -1.22665  0.272834  ... -0.498871 -0.613057  4.82861
 2.91671  0.387103 -1.94889  2.53074  ... -2.24878 -1.75281  9.31515
 3.95669  0.192163 -3.46541  3.02627  ... -3.97344 -2.93697 17.9105
 0.169872  0.713552  1.0551  -0.51381  ... -0.707596 -0.37273 -1.24571
-0.195013 -0.357777  2.08248 -1.67915  ...  1.21792 -0.293685 -7.44138
-4.50681 -0.0580125  3.68811 -4.20034  ...  4.21895  3.2872 -20.2689
-0.52739 -0.808059  1.88293 -0.459739  ...  0.183065 -0.159979 -4.93453
```

- `inv(A)`

## Control structures

Conditional execution:

```
cond1 = true
```

- `cond1=true`

```
cond2 = true
```

- `cond2=true`

```
"cond1"
```

- `if cond1`
- `"cond1"`
- `elseif cond2`
- `"cond2"`
- `else`
- `"nothing"`
- `end`

'?' operator for writing shorter code (borrowed from C):

```
"cond1"
```

- `cond1 ? "cond1" : "nothing"`

For loop:

```
1
2
3
4
5
```

- `with_terminal() do`
- `for i in 1:5`
- `println(i)`
- `end`
- `end`

Preliminary exit of a loop:

```
1
2
3
4
5
6
```

```
• with_terminal() do
•   for i in 1:10
•     println(i)
•     if i>5
•       break
•     end
•   end
• end
•
```

Skipping iterations:

```
1
2
3
4
6
7
8
9
10
```

```
• with_terminal() do
•   for i in 1:10
•     if i==5
•       continue
•     end
•     println(i)
•   end
• end
•
```

# Functions

- All arguments to functions are passed by reference
- Function name ending with ! indicates that the function mutates at least one argument, typically the first. This is a convention, not a syntax rule.
- Function objects can be assigned to variables

## Structure of function definition

```
function func(req1, req2, opt1=dflt1, opt2=dflt2; kw1=dflt3, kw2=dflt4)
    # do stuff
    return out1, out2, out3
end
```

- Required arguments are separated with a comma and use the positional notation
- Optional arguments have a default value in the signature and are positional
- Keyword arguments follow after the ; and have default values as well, they can be invoked in arbitrary sequence
- Return statement is optional, by default, the result of the last statement is returned
- Multiple outputs can be returned as a tuple, e.g., return out1, out2, out3.
- Return nothing if you would like to avoid returning data

## A function with one required and two optional arguments:

```
func_with_optional_args (generic function with 3 methods)
```

```
• function func_with_optional_args(x,y=9,z=9)
•     100*x+10*y+z
• end
```

199

```
• func_with_optional_args(1)
```

159

```
• func_with_optional_args(1,5)
```

178

```
• func_with_optional_args(1,7,8)
```

## A function with one required and two keyword arguments:

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

```
• function func_with_keyword_args(x;y=9,z=9)
•     100*x+10*y+z
• end
```

190

- `func_with_keyword_args(1; z=0)`

109

- `func_with_keyword_args(1; y=0)`

One line function definition:

`g` (generic function with 1 method)

- `g(x)=exp(sin(x))`

1.151562836514535

- `g(3)`

Nested function definitions:

`outerfunction` (generic function with 1 method)

- `function outerfunction(n)`
- `function innerfunction(i)`
- `println(i)`
- `end`
- `for i=1:n`
- `innerfunction(i)`
- `end`
- `end`

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

- `with_terminal() do`
- `outerfunction(13)`
- `end`

```

1
2
3
4
5
6
7
8
9
10
11
12
13

```

- `with_terminal(()->outerfunction(13))`

Functions are variables, too:

```
1.151562836514535
```

- `h=g; h(3)`

Functions as function parameters:

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

- `F(f,x)= f(x)`

```
1.151562836514535
```

- `F(g,3)`

Anonymous functions (convenient in function parameters):

```
0.1411200080598672
```

- `F(x -> sin(x),3)`

Do-block syntax: the body of first parameter is in the `do ... end` block:

```
1.151562836514535
```

- `F(3) do x`
- `exp(sin(x))`
- `end`

## Functions and vectors

Dot syntax can be used to make any function work on vectors:

```
v11 = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

- `v11=collect(0:0.1:1)`



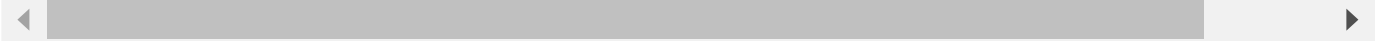
```
[1.0, 1.10499, 1.21978, 1.34383, 1.47612, 1.61515, 1.75882, 1.9045, 2.04901, 2.18874,
```



- `h.(v11)`

Map function on vector:

```
[1.0, 1.10499, 1.21978, 1.34383, 1.47612, 1.61515, 1.75882, 1.9045, 2.04901, 2.18874,
```



- `map(h, v11)`

`mapreduce`: apply operator to each element and collect data

```
0.0
```

- `mapreduce(x->x, *, v11)`

```
0.0
```

- `prod(v11)`

```
5.5
```

- `mapreduce(x->x, +, v11)`

```
5.5
```

- `sum(v11)`

## Macros

Julia allows to define macros which allow to modify Julia statements before they are compiled and executed. This capability is similar to the preprocessor in C or C++. Macro names start with `@`. Occasionally we will use predefined macros, e.g. `@elapsed` for returning the time used by some statement.

```
0.010412148
```

- `@elapsed inv(rand(100,100))`

The `@time` macro prints time and the number of allocation used for a statement:

```
100×100 Matrix{Float64}:
```

```
-0.56702   -0.402579   0.00589793 ... -0.42033   0.00302938   0.73175
-0.751877   0.379622   -1.34321   ... -1.81854   -0.24873   -1.84265
 0.133465  -0.0081091  0.427724   ...  0.807166   0.194797   0.307353
-0.341899   0.0485327  -0.52252   ... -0.722717   0.0361155  -0.303082
-0.760981   0.0900256  -0.795839   ... -1.38484   -0.16181   -0.638293
 0.303141  -0.119322   0.734361   ...  0.830899   0.0890981   0.71084
 1.24719   -0.0853247  1.13892   ...  2.06865   0.245249   0.971571
  ⋮
 2.10328   -0.482717   2.18122   ...  3.82102   0.838512   2.85292
 1.62697   -0.0472051  1.4893   ...  2.70135   0.362735   1.32342
 0.470655  -0.186648   0.610878   ...  0.921709   0.116817   0.777517
 0.0281604  0.331527   0.0142217  ... -0.130697  -0.0180337  -0.223652
 0.57301   -0.488904   1.09559   ...  1.79857   0.106815   1.11459
-0.361584   0.0310015  -0.213676  ... -0.303475  -0.104911  -0.390323
```

```
0.000218 seconds (7 allocations: 207.344 KiB)
```

```
• with_terminal() do
  • @time inv(rand(100,100))
  • end
```

```
• using PlutoUI
```