Julia - some basics

Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 02
 Jürgen Fuhrmann

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
- Types can be enforced by annotating variable definitions
- The `typeof` function allows to detect the type of a variable
- The `sizeof` function detects the size in bytes of a variable (1 byte = 8 bit)

Integers

Integer variables are used to represent integer numbers.

```
i = 10
```

```julia
i=10
```

```
Int64
```

```julia
typeof(i)
```

```
8
```

```julia
sizeof(i)
```

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

```
10
```

```julia
j::Int32=10
```

```
Int32
```

```julia
typeof(j)
```

```
4
```

```julia
sizeof(j)
```

Floating point numbers

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

```
x = 10.0
```

```julia
x=10.0
```

```
Float64
```

```julia
typeof(x)
```

```
8
```

```julia
sizeof(x)
```

There is also `Float32`. Both `Float64` and `Float32` follow the IEEE 754 standard floating point representation and are supported by typical computer hardware. There are also the slower, software implemented `Float16` and `BigFloat`.

```
y = 20.61428571428571590981196745165756770542689732142857142857142857142857142857135
```

```julia
y=BigFloat(144.3)/BigFloat(7)
```

```
40
```

```julia
sizeof(y)
```

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
- Indices count from 1!

Vector construction by explicit list of elements

```
v1 = [1, 2, 3, 4, 5, 6]
```

```
1 v1=[1,2,3,4,5,6]
```

```
Int64
```

```
1 eltype(v1)
```

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

```
1 typeof(v1)
```

```
48
```

```
1 sizeof(v1)
```

We can create a vector of floats:

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

```
1 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]
```

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

Other Vector constructors

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

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

```
1 zeros(Float32,5)
```

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

```
1 ones(5)
```

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

```
1 fill(17.0,10)
```

```
[0.79, 0.08057, 0.3423, 0.5728, 0.9985, 0.539, 0.527, 0.8984, 0.5684, 0.678, 0.1758, 0.34
```

```
1 rand(Float16,20)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 6.0f-45, 0.0]
```

```
1 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 = 1.0:0.5:10.0
```

```
1 r1=1:0.5:10
```

```
StepRangeLen{Float64, Base.TwicePrecision{Float64}, Base.TwicePrecision{Float64}, Int6
```

```
1 typeof(r1)
```

Collect the sequence from a range into a vector:

```
w1 =
▸ [1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 1
```

```
1 w1=collect(r1)
```

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, 0.973848
```

```
1 [sin(i) for i=1:0.1:5]
```

Vector dimensions

```
v6 = ▸ [1, 3, 5, 7, 9]
```

```
1 v6=collect(1:2:10)
```

`size` is a tuple of dimensions

```
▸ (5)
```

```
1 size(v6)
```

`length` describes the overall length:

```
5
```

```
1 length(v6)
```

Copies of parts of vectors

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

```
1 v7=collect(1:10)
```

```
subv7 = ▸ [2, 3, 4]
```

```
1 subv7=v7[2:4]
```

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

```
1 subv7[1]=17;v7
```

Views of parts of vectors

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

```
1 v8=collect(1:10)
```

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

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

```
▸ [1, 20, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
1 subv8[1]=20;v8
```

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

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

```
1 v9=collect(1:10)
```

```
@views subv9 = 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, 2.82743,
```

```
v10=collect(0:0.1π:2π)
```

```
[0.0, 0.309017, 0.587785, 0.809017, 0.951057, 1.0, 0.951057, 0.809017, 0.587785, 0.30901
```

```
sin.(v10)
```

```
[100.0, 100.314, 100.628, 100.942, 101.257, 101.571, 101.885, 102.199, 102.513, 102.827,
```

```
v10.+100
```

Matrices

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

```
(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  17
```

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

`undef` initialization:

```
3×3 Matrix{Float64}:
 2.0e-323  5.0e-324  6.91533e-310
 1.5e-323  2.5e-323  6.91533e-310
 1.0e-323  3.5e-323  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)
```

Linear Algebra

```
using LinearAlgebra
```

Let us create some linear algebra objects:

```
n = 7
```

```
n=7
```

```
w = [0.0289301, 0.309996, 0.370353, 0.956973, 0.0786183, 0.286121, 0.990613]
```

```
w=rand(n)
```

```
u = [0.172828, 0.655444, 0.931523, 0.882896, 0.775109, 0.343332, 0.167797]
```

```
u=rand(n)
```

```
A = 7×7 Matrix{Float64}:
 0.122273   0.143389   0.280711  0.055371  0.745358  0.981165  0.573053
 0.0298344  0.427139   0.769754  0.825182  0.654736  0.402289  0.169028
 0.63263    0.751614   0.49709   0.295618  0.656714  0.24233   0.837514
 0.575123   0.361328   0.586174  0.414048  0.101159  0.484291  0.00768385
 0.970395   0.032978   0.557866  0.814976  0.111174  0.263422  0.625582
 0.266219   0.409322   0.218136  0.319655  0.463038  0.487139  0.886449
 0.217392   0.0822313  0.237288  0.319027  0.449496  0.186591  0.183085
```

```
A=rand(n,n)
```

Some operations

Mean square norm $||u||_2 = \sqrt{\sum_{i=1}^n u_i^2}$:

```
1.6892435085357238
```

```
norm(u)
```

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

```
1.7234777891590158
```

```
dot(u,w)
```

Matrix-vector product Au:

```
[1.43625, 2.40469, 2.05879, 1.4938, 1.71011, 1.47462, 1.03737]
```

```
A*u
```

Inverse and \

```
7×7 Matrix{Float64}:
 -0.439904   -1.37751    0.0557499   1.0831    -0.0983358   0.0653549   2.36774
 -1.63767    -0.582633  -0.321908    1.92474   -1.70955     2.32781     1.62629
  2.63586     2.39253    2.52852    -2.4843     2.20773    -4.70354    -6.69163
 -1.88708    -0.265321  -1.99681     1.17701   -0.699408    2.96073     3.29116
  0.0747831  -0.351476   0.332705   -0.325743  -0.521749   -0.515275    2.85974
  0.138876   -0.535622  -1.26067     1.4374    -0.590816    1.46152     0.708833
  0.804789    0.667594   0.748741   -1.64689    1.12507    -0.410618   -2.88548
```

```
inv(A)
```

```
b = [0.929252, 0.349782, 0.298991, 0.34435, 0.34535, 0.696165, 0.574313]
```

```
b=rand(n)
```

Solve $Ax = b$:

▸ [0.870382, 0.80508, -3.16832, 1.67167, 1.03734, 1.28025, -0.916359]

```
A\(b)
```

▸ [0.870382, 0.80508, -3.16832, 1.67167, 1.03734, 1.28025, -0.916359]

```
inv(A)*b
```

Conditional execution

```
cond1 = true
```

```
cond1=true
```

```
cond2 = true
```

```
cond2=true
```

"cond1"

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

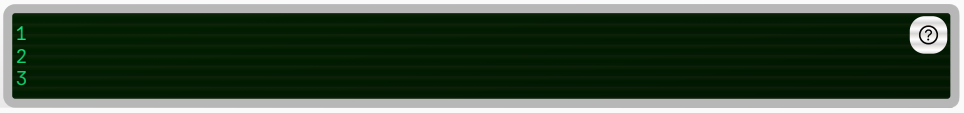
'?' operator for writing shorter code (borrowed from C):

"cond1"

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

For loop

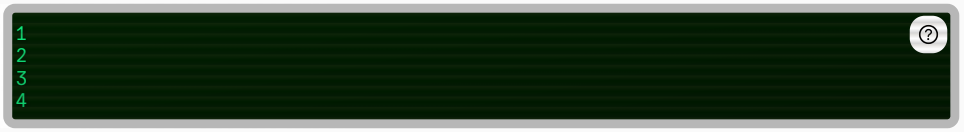
```
for i ∈ 1:3
        println(i)
end
```



Preliminary exit of a loop:

```
for i in 1:5
        println(i)
        if i>3
            break
        end
end

```



Skipping iterations:

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

Function with one requiered, two optional args:

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

```
1 function func_with_optional_args(x,y=9,z=9)
2     100*x+10*y+z
3 end
```

```
199
```

```
1 func_with_optional_args(1)
```

```
159
```

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

```
178
```

```
1 func_with_optional_args(1,7,8)
2
```

Function with one required and 2 keyword args:

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

```
1 function func_with_keyword_args(x;y=9,z=9)
2     100*x+10*y+z
3 end
```

```
199
```

```
1 func_with_keyword_args(1)
```

```
190
```

```
1 func_with_keyword_args(1; z=0)
```

```
109
```

```
1 func_with_keyword_args(1; y=0)
```

Passing keyword arguments

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

```
function func_passing_keyword_args(a; kwargs...)
    10*func_with_keyword_args(a;kwargs...)
end
```

```
1990
```

```
func_passing_keyword_args(1)
```

```
1900
```

```
func_passing_keyword_args(1; z=0)

```

Nested function definitions

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

```
function outerfunction(n)
    function innerfunction(i)
        println("inner: $i")
    end
    for i=1:n
        innerfunction(i)
    end
end
```

```
outerfunction(10)
```

```
inner: 1
inner: 2
inner: 3
inner: 4
inner: 5
inner: 6
inner: 7
inner: 8
inner: 9
inner: 10
```

One line function definition

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

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

```
1.151562836514535
```

```
g(3)
```

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

These are convenient for function parameters.

```
0.1411200080598672
```

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

Do-block syntax

The body of the 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, 2.3
```

```
h.(v11)
```

Higher order functional programming

Map function on vector:

```
[1.0, 1.10499, 1.21978, 1.34383, 1.47612, 1.61515, 1.75882, 1.9045, 2.04901, 2.18874, 2.3
```

```
map(h,v11)
```

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

```
5.5
```

```
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.

```
0.000426348
```

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

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

```
100×100 Matrix{Float64}:
 -0.975186     0.494867     0.394954    …   3.18128    -3.05795    -1.15569
  1.10148     -0.396712    -0.121272       -4.51354     4.03856     1.37606
  0.628216    -0.202754    -0.0980053      -2.07038     1.4579      0.399876
 -0.00706263   0.144198    -0.0179404      -0.0867205   0.0918086  -0.0293865
 -0.0797674    0.0305559   -0.00613314     -0.0287555   0.382855    0.259971
 -0.267921     0.590776     0.00551646  …   0.163651    0.1077      0.318051
 -0.315829    -0.290215     0.202995        2.15148    -2.62213    -0.905404
  ⋮                                     ⋱
  0.36098      0.351945     0.118369       -2.43004     2.13578     1.14974
 -0.426628     0.243768    -0.17157    …   1.48227    -0.297472   -0.441351
 -1.23514     -0.0599416    0.293574        4.11433    -2.91483    -0.788939
  0.719728    -0.366517    -0.0442644      -3.99991     3.97697     1.68864
 -0.614188    -0.00778489   0.359038        2.52969    -2.36245    -0.756767
  0.58076     -0.274369    -0.233724       -1.71681     1.42161     0.300068
```

```
1 @time inv(rand(100,100))
```

```
  0.000292 seconds (12 allocations: 207.406 KiB)
```