# The Julia version 0.5 Express

**Bogumił Kamiński (http://bogumilkaminski.pl)**

## Contents



## Inroduction

The Purpose of this document is to introduce programmers to Julia programming by example. This is a simplified exposition of the language.

If some packages are missing on your system use Pkg.add to require installing them. There are many add-on packages which you can browse at http://pkg.julialang.org/.

Major stuff not covered (please see the documentation):

1. parametric types;
2. parallel and distributed processing;
3. advanced I/O operations;
4. package management; see `Pkg`;
5. interaction with system shell; see `run`;
6. exception handling; see `try`;
7. creation of coroutines; see `Task`;
8. two-way integration with C and Fortran.

You can find current Julia documentation at http://julia.readthedocs.org/en/latest/manual/.

The code was executed using the following Julia version:

In [2]:
```
versioninfo()
```

```
Julia Version 1.7.1
Commit ac5cc99908 (2021-12-22 19:35 UTC)
Platform Info:
  OS: macOS (x86_64-apple-darwin19.5.0)
  CPU: Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-12.0.1 (ORCJIT, ivybridge)
```

Remember that you can expect every major version of Julia to introduce breaking changes.

Check https://github.com/JuliaLang/julia/blob/master/NEWS.md for release notes.

All sugestions how this guide can be improved are welcomed. Please contact me at bkamins@sgh.waw.pl.

## Getting around

Running julia invokes interactive (REPL) mode. In this mode some useful commands are:

1. ^D (exits Julia, also calling `exit(c)` quits with exit code c);
2. ^C (interrupts computations);
3. ? (enters help mode)
4. ; (enters system shell mode)
5. putting ; after the expression will disable showing of its value.

Examples of some essential functions in REPL (they can be also invoked in scripts):

In [3]:
```
apropos("apropos") # search documentation for "apropos" string
```

```
Base.Docs.apropos
Base.Docs.stripmd
```

In [4]:
```
@less max(1,2) # show the definition of max function when invoked with arguments 1 and 2
```

```
min(x::T, y::T) where {T<:Real} = select_value(y < x, y, x)
minmax(x::T, y::T) where {T<:Real} = y < x ? (y, x) : (x, y)
```

In [5]:
```
whos() # list of global variables and their types
```

```
                         Base                  Module
                       Compat  21304 KB        Module
                         Core                  Module
                       IJulia  21506 KB        Module
                         JSON  21416 KB        Module
                         Main                  Module
                      MbedTLS  21425 KB        Module
                          ZMQ  21366 KB        Module
```

In [6]:
```
cd("D:/") # change working to D:/ (on Windows)
pwd() # get current working directory
```

Out [6]:
```
"D:\\"
```

In [7]:
```
include("file.jl") # execute source file, LoadError if execution fails
```

```
could not open file D:\file.jl

Stacktrace:
 [1] include_from_node1(::String) at .\loading.jl:552
 [2] include(::String) at .\sysimg.jl:14
```

In [8]:
```
clipboard([1,2]) # copy data to system clipboard
clipboard() # load data from system clipboard as string
```

Out [8]:
```
"[1, 2]"
```

In [9]:
```
workspace() # clear worskspace - create new Main module (only to be used interactively)
```

You can execute Julia script by running `julia script.jl`.

Try saving the following example script to a file and run it (more examples of all the constructs used are given in following sections):

In [10]:
```
"Sieve of Eratosthenes function docstring"
function es(n::Int) # accepts one integer argument
    isprime = ones(Bool, n) # n-element vector of true-s
    isprime[1] = false # 1 is not a prime
    for i in 2:round(Int, sqrt(n)) # loop integers from 2 to sqrt(n)
        if isprime[i] # conditional evaluation
            for j in (i*i):i:n # sequence with step i
                isprime[j] = false
            end
        end
    end
    return filter(x -> isprime[x], 1:n) # filter using anonymous function
end
println(es(100)) # print all primes less or equal than 100
@time length(es(10^7)) # check function execution time and memory usage
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
  0.136820 seconds (281 allocations: 24.162 MiB, 4.81% gc time)
```

Out [10]: 664579

## Basic literals and types

Basic scalar literals (`x::Type` is a literal `x` with type `Type` assertion):

In [11]:
```
1::Int64 # 64-bit integer, no overflow warnings, fails on 32 bit Julia
```

Out [11]: 1

In [12]:
```
1.0::Float64 # 64-bit float, defines NaN, -Inf, Inf
```

Out [12]: 1.0

In [13]:
```
true::Bool # boolean, allows "true" and "false"
```

Out [13]: true

In [14]:
```
'c'::Char # character, allows Unicode
```

Out [14]: 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)

In [15]:
```
"s"::AbstractString # strings, allows Unicode, see also Strings
```

Out [15]: "s"

All basic types are immutable. Specifying type assertion is optional (and usually it is not needed, but I give it to show how you can do it). Type assertions for variables are made in the same way and may improve code performance.

If you do not specify type assertion Julia will choose a default. Note that defaults might be different on 32-bit and 64-bit versions of Julia. A most important difference is for integers which are `Int32` and `Int64` respectively. This means that `1::Int32` assertion will fail on 64-bit version. Notably `Int` is either `Int64` or `Int32` depending on version (the same with `UInt`). There is no automatic type conversion (especially important in function calls). Has to be explicit:

In [16]:
```
Int64('a') # character to integer
```

Out [16]: 97

In [17]:
```
Int64(2.0) # float to integer
```

Out [17]: 2

In [18]:
```
Int64(1.3) # inexact error
```

```
InexactError()

Stacktrace:
 [1] convert(::Type{Int64}, ::Float64) at .\float.jl:679
 [2] Int64(::Float64) at .\sysimg.jl:24
```

In [19]:
```
Int64("a") # error no conversion possible
```

```
MethodError: Cannot `convert` an object of type String to an object of type Int64
This may have arisen from a call to the constructor Int64(...),
since type constructors fall back to convert methods.

Stacktrace:
 [1] Int64(::String) at .\sysimg.jl:24
```

In [20]:
```
Float64(1) # integer to float
```

Out [20]: 1.0

In [21]:
```
Bool(1) # converts to boolean true
```

Out [21]: true

In [22]:
```
Bool(0) # converts to boolean false
```

Out [22]: false

In [23]:
```
Bool(2) # conversion error
```

```
InexactError()

Stacktrace:
 [1] convert(::Type{Bool}, ::Int64) at .\bool.jl:7
 [2] Bool(::Int64) at .\sysimg.jl:24
```

In [24]:
```
Char(89) # integer to char
```

Out [24]:
```
'Y': ASCII/Unicode U+0059 (category Lu: Letter, uppercase)
```

In [25]:
```
string(true) # cast bool to string (works with other types, note small caps)
```

Out [25]:
```
"true"
```

In [26]:
```
string(1,true) # string can take more than one argument and concatenate them
```

Out [26]:
```
"1true"
```

In [27]:
```
zero(10.0) # zero of type of 10.0
```

Out [27]:
```
0.0
```

In [28]:
```
one(Int64) # one of type Int64
```

Out [28]:
```
1
```

General conversion can be done using `convert(Type, x)`:

In [29]:
```
convert(Int64, 1.0) # convert float to integer
```

Out [29]:
```
1
```

Parsing strings can be done using `parse(Type, str)`:

In [30]:
```
parse(Int64, "1") # parse "1" string as Int64
```

Out [30]:
```
1
```

Automatic promotion ofmany arguments to common type (if any) using `promote`:

In [31]:
```
promote(true, BigInt(1)//3, 1.0) # tuple (see Tuples) of BigFloats, true promoted to 1.0
```

Out [31]:
```
(1.0000000000000000000000000000000000000000000000000000000000000000000000000000,
3.33333333333333333333333333333333333333333333333333333333333333333333333333348e-01,
1.0000000000000000000000000000000000000000000000000000000000000000000000000000)
```

In [32]:
```
promote("a", 1) # promotion to common type not possible
```

Out [32]:
```
("a", 1)
```

Many operations (arithmetic, assignment) are defined in a way that performs automatic type promotion. One can verify type of argument:

In [33]:
```
typeof("abc") # String returned which is a AbstractString subtype
```

Out [33]:
```
String
```

In [34]:
```
isa("abc", AbstractString) # true
```

Out [34]:
```
true
```

In [35]:
```
isa(1, Float64) # false, integer is not a float
```

Out [35]:
```
false
```

In [36]:
```
isa(1.0, Float64) # true
```

Out [36]:
```
true
```

In [37]:
```
isa(1.0, Number) # true, Number is abstract type
```

Out [37]: true

In [38]:
```julia
supertype(Int64) # supertype of Int64
```

Out [38]: Signed

In [39]:
```julia
subtypes(Real) # subtypes of bastract type Real
```

Out [39]:
```
4-element Array{Union{DataType, UnionAll},1}:
 AbstractFloat
 Integer
 Irrational
 Rational
```

In [40]:
```julia
subtypes(Int64)
```

Out [40]: 0-element Array{Union{DataType, UnionAll},1}

It is possible to performcalculations using arbitrary precision arithmetic or rational numbers:

In [41]:
```julia
BigInt(10)^1000 # big integer
```

Out [41]: 1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

In [42]:
```julia
BigFloat(10)^1000 # big float, see documentation how to change default precision
```

Out [42]: 1.00000000000000000000000000000000000000000000000000000000000000000000000000004e+1000

In [43]:
```julia
123//456 # rational numbers are created using // operator
```

Out [43]: 41//152

Type hierarchy of all standard numeric types is given below:

In [44]:
```julia
function type_hierarchy(t::DataType, level = 0)
    println(" "^level, t)
    for x in subtypes(t)
        type_hierarchy(x, level+2)
    end
end
type_hierarchy(Number)
```

```
Number

MethodError: no method matching type_hierarchy(::Type{Complex}, ::Int64)
Closest candidates are:
  type_hierarchy(::DataType, ::Any) at In[44]:2

Stacktrace:
 [1] type_hierarchy(::DataType, ::Int64) at .\In[44]:4
 [2] type_hierarchy(::DataType) at .\In[44]:2
```

## Complex literals and types

Important types:

In [45]:
```julia
Any # all objects are of this type
```

Out [45]: Any

In [46]:
```julia
Union{} # subtype of all types, no object can have this type
```

Out [46]: Union{}

In [47]:
```julia
Void # type indicating nothing, subtype of Any
```

Out [47]: Void

In [48]:
```julia
nothing # only instance of Void
```

Additionally `#undef` indicates an incompletely initialized instance.

### Tuples

Tuples are immutable sequences indexed from `1`:

In [49]:
```
() # empty tuple
```

Out [49]:
```
()
```

In [50]:
```
(1,) # one element tuple
```

Out [50]:
```
(1,)
```

In [51]:
```
("a", 1) # two element tuple
```

Out [51]:
```
("a", 1)
```

In [52]:
```
('a', false)::Tuple{Char, Bool} # tuple type assertion
```

Out [52]:
```
('a', false)
```

In [53]:
```
x = (1, 2, 3)
```

Out [53]:
```
(1, 2, 3)
```

In [54]:
```
x[1] # 1 (element)
```

Out [54]:
```
1
```

In [55]:
```
x[1:2] # (1, 2) (tuple)
```

Out [55]:
```
(1, 2)
```

In [56]:
```
x[4] # bounds error
```

```
BoundsError: attempt to access (1, 2, 3)
  at index [4]

Stacktrace:
 [1] getindex(::Tuple{Int64,Int64,Int64}, ::Int64) at .\tuple.jl:21
```

In [57]:
```
x[1] = 1 # error - tuple is not mutable
```

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

In [58]:
```
a, b = x # tuple unpacking a=1, b=2
println("$a $b")
```

```
1 2
```

### Arrays

Arrays are mutable and passed by reference. Array creation:

In [59]:
```
Array{Char}(2, 3, 4) # 2x3x4 array of Chars
```

Out [59]:
```
2×3×4 Array{Char,3}:
[:, :, 1] =
 '\U15fec730'  '\U15fec770'  '\U15fec7d0'
 '\0'          '\0'          '\0'

[:, :, 2] =
 '\U15fec810'  '\U15fec850'  '\U15fec890'
 '\0'          '\0'          '\0'

[:, :, 3] =
 '\U15feca90'  '\U15fec610'  '\U15fec9d0'
 '\0'          '\0'          '\0'

[:, :, 4] =
 '\U15fed1d0'  '\U16105830'  '\U161058f0'
```

```
 '\0'          '\0'          '\0'
```

In [60]:
```
Array{Int64}(0, 0) # degenerate 0x0 array of Int64
```

Out [60]:
```
0×0 Array{Int64,2}
```

In [61]:
```
Array{Any}(2, 3) # 2x3 array of Any
```

Out [61]:
```
2×3 Array{Any,2}:
 #undef  #undef  #undef
 #undef  #undef  #undef
```

In [62]:
```
zeros(5) # vector of Float64 zeros
```

Out [62]:
```
5-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0
 0.0
```

In [63]:
```
ones(5) # vector of Float64 ones
```

Out [63]:
```
5-element Array{Float64,1}:
 1.0
 1.0
 1.0
 1.0
 1.0
```

In [64]:
```
ones(Int64, 2, 1) # 2x1 array of Int64 ones
```

Out [64]:
```
2×1 Array{Int64,2}:
 1
 1
```

In [65]:
```
trues(3), falses(3) # tuple of vector of trues and of falses
```

Out [65]:
```
(Bool[true, true, true], Bool[false, false, false])
```

In [66]:
```
eye(3) # 3x3 Float64 identity matrix
```

Out [66]:
```
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

In [67]:
```
x = linspace(1, 2, 5) # iterator having 5 equally spaced elements
```

Out [67]:
```
1.0:0.25:2.0
```

In [68]:
```
collect(x) # converts iterator to vector
```

Out [68]:
```
5-element Array{Float64,1}:
 1.0
 1.25
 1.5
 1.75
 2.0
```

In [69]:
```
1:10 # iterable from 1 to 10
```

Out [69]:
```
1:10
```

In [70]:
```
1:2:10 # iterable from 1 to 9 with 2 skip
```

Out [70]:
```
1:2:9
```

In [71]:
```
reshape(1:12, 3, 4) # 3x4 array filled with 1:12 values
```

Out [71]:
```
3×4 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
 1  4  7  10
 2  5  8  11
 3  6  9  12
```

In [72]:
```
fill("a", 2, 2) # 2x2 array filled with "a"
```

```
Out [72]: 2×2 Array{String,2}:
 "a"  "a"
 "a"  "a"
```

```
In [73]: repmat(eye(2), 3, 2) # 2x2 identity matrix repeated 3x2 times
```

```
Out [73]: 6×4 Array{Float64,2}:
 1.0  0.0  1.0  0.0
 0.0  1.0  0.0  1.0
 1.0  0.0  1.0  0.0
 0.0  1.0  0.0  1.0
 1.0  0.0  1.0  0.0
 0.0  1.0  0.0  1.0
```

```
In [74]: x = [1, 2] # two element vector !!!!!!!! add x' here!
```

```
Out [74]: 2-element Array{Int64,1}:
 1
 2
```

```
In [75]: resize!(x, 5) # resize x in place to hold 5 values (filled with garbage)
```

```
Out [75]: 5-element Array{Int64,1}:
      1
      2
 569348
      0
      0
```

```
In [76]: [1] # vector with one element (not a scalar)
```

```
Out [76]: 1-element Array{Int64,1}:
 1
```

```
In [77]: [x * y for x in 1:2, y in 1:3] # comprehension generating 2x3 array
```

```
Out [77]: 2×3 Array{Int64,2}:
 1  2  3
 2  4  6
```

```
In [78]: Float64[x^2 for x in 1:4] # casting comprehension result to Float64
```

```
Out [78]: 4-element Array{Float64,1}:
  1.0
  4.0
  9.0
 16.0
```

```
In [79]: [1 2] # 1x2 matrix (hcat function)
```

```
Out [79]: 1×2 Array{Int64,2}:
 1  2
```

```
In [80]: [1 2]' # 2x1 matrix (after transposing)
```

```
Out [80]: 2×1 Array{Int64,2}:
 1
 2
```

```
In [81]: [1, 2] # vector (concatenation)
```

```
Out [81]: 2-element Array{Int64,1}:
 1
 2
```

```
In [82]: [1; 2] # vector (vcat function)
```

```
Out [82]: 2-element Array{Int64,1}:
 1
 2
```

```
In [83]: [1 2 3; 1 2 3] # 2x3 matrix (hvcat function)
```

```
Out [83]: 2×3 Array{Int64,2}:
 1  2  3
 1  2  3
```

In [84]:
```
[1; 2] == [1 2]' # false, different array dimensions
```
Out [84]:
```
false
```
In [85]:
```
[(1, 2)] # 1-element vector
```
Out [85]:
```
1-element Array{Tuple{Int64,Int64},1}:
 (1, 2)
```
In [86]:
```
collect((1, 2)) # 2-element vector by tuple unpacking
```
Out [86]:
```
2-element Array{Int64,1}:
 1
 2
```
In [87]:
```
[[1 2] 3] # append to a row vector (hcat)
```
Out [87]:
```
1×3 Array{Int64,2}:
 1  2  3
```
In [88]:
```
[[1; 2]; 3] # append to a column vector (vcat)
```
Out [88]:
```
3-element Array{Int64,1}:
 1
 2
 3
```
Vectors (1D arrays) are treated as column vectors.

Julia offers sparse and distributed matrices (see documentation for details).

Commonly needed array utility functions:

In [89]:
```
a = [x * y for x in 1:2, y in 1, z in 1:3] # 2x3 array of Int64; singelton dimension is dropped
```
Out [89]:
```
2×3 Array{Int64,2}:
 1  1  1
 2  2  2
```
In [90]:
```
a = [x * y for x in 1:2, y in 1:1, z in 1:3] # 2x1x3 array of Int64; singelton dimension is not dr
```
Out [90]:
```
2×1×3 Array{Int64,3}:
[:, :, 1] =
 1
 2

[:, :, 2] =
 1
 2

[:, :, 3] =
 1
 2
```
In [91]:
```
ndims(a) # number of dimensions in a
```
Out [91]:
```
3
```
In [92]:
```
eltype(a) # type of elements in a
```
Out [92]:
```
Int64
```
In [93]:
```
length(a) # number of elements in a
```
Out [93]:
```
6
```
In [94]:
```
size(a) # tuple containing dimension sizes of a
```
Out [94]:
```
(2, 1, 3)
```
In [95]:
```
vec(a) # cast array to vetor (single dimension)
```
Out [95]:
```
6-element Array{Int64,1}:
 1
 2
 1
```

```
 2
 1
 2
```

In [96]:
```
squeeze(a, 2) # remove 2nd dimension as it has size 1
```

Out [96]:
```
2×3 Array{Int64,2}:
 1  1  1
 2  2  2
```

In [97]:
```
sum(a, 3) # calculate sums for 3rd dimensions, similarly: mean, std, prod, minimum, maximum, any,
```

Out [97]:
```
2×1×1 Array{Int64,3}:
[:, :, 1] =
 3
 6
```

In [98]:
```
count(x -> x > 0, a) # count number of times a predicate is true, similar: all, any
```

Out [98]:
```
6
```

Array access functions:

In [99]:
```
a = linspace(0, 1) # LinSpace{Float64} of length 50
```

Out [99]:
```
0.0:0.02040816326530612:1.0
```

In [100]:
```
a[1] # get scalar 0.0
```

Out [100]:
```
0.0
```

In [101]:
```
a[end] # get scalar 1.0 (last position)
```

Out [101]:
```
1.0
```

In [102]:
```
a[1:2:end] # every second element from range, LinSpace{Float64}
```

Out [102]:
```
0.0:0.04081632653061224:0.9795918367346939
```

In [103]:
```
a[repmat([true, false], 25)] # select every second element, Array{Float64,1}
```

Out [103]:
```
25-element Array{Float64,1}:
 0.0
 0.0408163
 0.0816327
 0.122449
 0.163265
 0.204082
 0.244898
 0.285714
 0.326531
 0.367347
 0.408163
 0.44898
 0.489796
 0.530612
 0.571429
 0.612245
 0.653061
 0.693878
 0.734694
 0.77551
 0.816327
 0.857143
 0.897959
 0.938776
 0.979592
```

In [104]:
```
a[[1, 3, 6]] # 1st, 3rd and 6th element of a, Array{Float64,1}
```

Out [104]:
```
3-element Array{Float64,1}:
 0.0
 0.0408163
 0.102041
```

In [105]:
```
view(a, 1:2:50) # view into subsarray of a
```

Out [105]: 25-element
SubArray{Float64,1,StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}},T

```
 0.0
 0.0408163
 0.0816327
 0.122449
 0.163265
 0.204082
 0.244898
 0.285714
 0.326531
 0.367347
 0.408163
 0.44898
 0.489796
 0.530612
 0.571429
 0.612245
 0.653061
 0.693878
 0.734694
 0.77551
 0.816327
 0.857143
 0.897959
 0.938776
 0.979592
```

In [106]:
```
endof(a) # last index of the collection a
```

Out [106]: 50

Observe the treatment of trailing singleton dimensions:

In [107]:
```
a = reshape(1:12, 3, 4)
```

Out [107]:
```
3×4 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
 1  4  7  10
 2  5  8  11
 3  6  9  12
```

In [108]:
```
a[:, 1:2] # 3x2 matrix
```

Out [108]:
```
3×2 Array{Int64,2}:
 1  4
 2  5
 3  6
```

In [109]:
```
a[:, 1] # 3 element vector
```

Out [109]:
```
3-element Array{Int64,1}:
 1
 2
 3
```

In [110]:
```
a[1, :] # 4 element vector
```

Out [110]:
```
4-element Array{Int64,1}:
  1
  4
  7
 10
```

In [111]:
```
a[1:1, :] # 1x4 matrix
```

Out [111]:
```
1×4 Array{Int64,2}:
 1  4  7  10
```

In [112]:
```
a[:, :, 1, 1] # works 3x4 matrix
```

Out [112]:
```
3×4 Array{Int64,2}:
 1  4  7  10
 2  5  8  11
 3  6  9  12
```

In [113]:
```
a[:, :, :, [true]] # wroks 3x4x1 matrix
```

Out [113]: 3×4×1×1 Array{Int64,4}:

```
[:, :, 1, 1] =
 1  4  7  10
 2  5  8  11
 3  6  9  12
```

In [114]:
```julia
a[1, 1, [false]] # works 0-element Array{Int64,1}
```

Out [114]:
```
0-element Array{Int64,1}
```

In [115]:
```julia
a[] # first element of an array
```

Out [115]:
```
1
```

Array assignment:

In [116]:
```julia
x = collect(reshape(1:8, 2, 4))
```

Out [116]:
```
2×4 Array{Int64,2}:
 1  3  5  7
 2  4  6  8
```

In [117]:
```julia
x[:, 2:3] = [1 2] # error; size mismatch
```

```
DimensionMismatch("tried to assign 1×2 array to 2×2 destination")

Stacktrace:
 [1] throw_setindex_mismatch(::Array{Int64,2}, ::Tuple{Int64,Int64}) at .\indices.jl:94
 [2] setindex_shape_check(::Array{Int64,2}, ::Int64, ::Int64) at .\indices.jl:151
 [3] macro expansion at .\multidimensional.jl:501 [inlined]
 [4] _unsafe_setindex!(::IndexLinear, ::Array{Int64,2}, ::Array{Int64,2}, ::Base.Slice{Base.OneTo{In
 [5] macro expansion at .\multidimensional.jl:488 [inlined]
 [6] _setindex! at .\multidimensional.jl:484 [inlined]
 [7] setindex!(::Array{Int64,2}, ::Array{Int64,2}, ::Colon, ::UnitRange{Int64}) at .\abstractarray.j
```

In [118]:
```julia
x[:, 2:3] = repmat([1 2], 2) # OK
```

Out [118]:
```
2×2 Array{Int64,2}:
 1  2
 1  2
```

In [119]:
```julia
x[:, 2:3] = 3 # OK
```

Out [119]:
```
3
```

Arrays are assigned and passed by reference. Therefore copying is provided:

In [120]:
```julia
x = Array{Any}(2)
x[1] = ones(2)
x[2] = trues(3)
a = x
b = copy(x) # shallow copy
c = deepcopy(x) # deep copy
x[1] = "Bang"
x[2][1] = false
x
```

Out [120]:
```
2-element Array{Any,1}:
 "Bang"
 Bool[false, true, true]
```

In [121]:
```julia
a # identical as x
```

Out [121]:
```
2-element Array{Any,1}:
 "Bang"
 Bool[false, true, true]
```

In [122]:
```julia
b # only x[2][1] changed from original x
```

Out [122]:
```
2-element Array{Any,1}:
 [1.0, 1.0]
 Bool[false, true, true]
```

In [123]:
```julia
c # contents to original x
```

Out [123]:
```
2-element Array{Any,1}:
 [1.0, 1.0]
 Bool[true, true, true]
```

Array types syntax examples:

In [124]:
```
[1 2]::Array{Int64, 2} # 2 dimensional array of Int64
```

Out [124]:
```
1×2 Array{Int64,2}:
 1  2
```

In [125]:
```
[true; false]::Vector{Bool} # vector of Bool
```

Out [125]:
```
2-element Array{Bool,1}:
  true
 false
```

In [126]:
```
[1 2; 3 4]::Matrix{Int64} # matrix of Int64
```

Out [126]:
```
2×2 Array{Int64,2}:
 1  2
 3  4
```

### Composite types

Composite types are mutable and passed by reference. You can define and access composite types:

In [127]:
```
mutable struct Point
    x::Int64
    y::Float64
    meta
end
p = Point(0, 0.0, "Origin")
```

Out [127]:
```
Point(0, 0.0, "Origin")
```

In [128]:
```
p.x # access field
```

Out [128]:
```
0
```

In [129]:
```
p.meta = 2 # change field value
```

Out [129]:
```
2
```

In [130]:
```
p.x = 1.5 # error, wrong data type
```

```
InexactError()

Stacktrace:
 [1] convert(::Type{Int64}, ::Float64) at .\float.jl:679
```

In [131]:
```
p.z = 1 # error - no such field
```

```
type Point has no field z
```

In [132]:
```
fieldnames(p) # get names of instance fields
```

Out [132]:
```
3-element Array{Symbol,1}:
 :x
 :y
 :meta
```

In [133]:
```
fieldnames(Point) # get names of type fields
```

Out [133]:
```
3-element Array{Symbol,1}:
 :x
 :y
 :meta
```

You can define type to be immutable by removing word `mutable`. There are also union types (see documentation for details).

### Dictionaries

Associative collections (key-value dictionaries):

In [134]:
```
x = Dict{Float64, Int64}() # empty dictionary mapping floats to integers
```

Out [134]:
```
Dict{Float64,Int64} with 0 entries
```

In [135]:
```
y = Dict("a"=>1, "b"=>2) # filled dictionary
```

Out [135]:
```
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1
```

In [136]:
```
y["a"] # element retrieval
```

Out [136]:
```
1
```

In [137]:
```
y["c"] # error
```

```
KeyError: key "c" not found

Stacktrace:
 [1] getindex(::Dict{String,Int64}, ::String) at .\dict.jl:474
```

In [138]:
```
y["c"] = 3 # added element
```

Out [138]:
```
3
```

In [139]:
```
haskey(y, "b") # check if y contains key "b"
```

Out [139]:
```
true
```

In [140]:
```
keys(y), values(y) # tuple of iterators returning keys and values in y
```

Out [140]:
```
(String["c", "b", "a"], [3, 2, 1])
```

In [141]:
```
delete!(y, "b") # delete key from a collection, see also: pop!
```

Out [141]:
```
Dict{String,Int64} with 2 entries:
  "c" => 3
  "a" => 1
```

In [142]:
```
get(y,"c","default") # return y["c"] or "default" if not haskey(y,"c")
```

Out [142]:
```
3
```

Julia also supports operations on sets and dequeues, priority queues and heaps (please refer to documentation).

## Strings

String operations:

In [143]:
```
"Hi " * "there!" # string concatenation
```

Out [143]:
```
"Hi there!"
```

In [144]:
```
"Ho " ^ 3 # repeat string
```

Out [144]:
```
"Ho Ho Ho "
```

In [145]:
```
string("a= ", 123.3) # create using print function
```

Out [145]:
```
"a= 123.3"
```

In [146]:
```
repr(123.3) # fetch value of show function to a string
```

Out [146]:
```
"123.3"
```

In [147]:
```
contains("ABCD", "CD") # check if first string contains second
```

Out [147]: true

In [148]:
```
"\"\n\t\$" # C-like escaping in strings, new \$ escape
```

Out [148]: "\"\n\t\$"

In [149]:
```
x = 123
"$x + 3 = $(x+3)" # unescaped $ is used for interpolation
```

Out [149]: "123 + 3 = 126"

In [150]:
```
"\$199" # to get a $ symbol you must escape it
```

Out [150]: "\$199"

PCRE regular expressions handling:

In [151]:
```
r = r"A|B" # create new regexp
```

Out [151]: r"A|B"

In [152]:
```
ismatch(r, "CD") # false, no match found
```

Out [152]: false

In [153]:
```
m = match(r, "ACBD") # find first regexp match, see documentation for details
```

Out [153]: RegexMatch("A")

There is a vast number of string functions—please refer to documentation.

## Programming constructs

The simplest way to create new variable is by assignment:

In [154]:
```
x = 1.0 # x is Float64
```

Out [154]: 1.0

In [155]:
```
x = 1 # now x is Int32 on 32 bit machine and Int64 on 64 bit machine
```

Out [155]: 1

Expressions can be compound using ; or begin end block:

In [156]:
```
x = (a = 1; 2 * a) # after: x = 2; a = 1
(x, a)
```

Out [156]: (2, 1)

In [157]:
```
y = begin
    b = 3
    3 * b
end # after: y = 9; b = 3
(y, b)
```

Out [157]: (9, 3)

There are standard programming constructs:

In [158]:
```
if false # if clause requires Bool test
    z = 1
elseif 1==2
    z = 2
else
    a = 3
end # after this a = 3 and z is undefined
(a, isdefined(:z))
```

Out [158]: (3, false)

In [159]:
```
1==2 ? "A" : "B" # standard ternary operator
```

Out [159]: "B"

In [160]:
```
i = 1
while true
    i += 1
    if i > 10
        break
    end
end
i
```

Out [160]: 11

In [161]:
```
for x in 1:10 # x in collection, can also use = here instead of in
    if 3 < x < 6
        continue # skip one iteration
    end
    println(x)
end # x is introduced in loop outer scope
x
```

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

Out [161]: 10

You can define your own functions:

In [162]:
```
f(x, y = 10) = x + y # new function f with y defaulting to 10; last result returned
f(3, 2) # simple call, 5 returned
```

Out [162]: 5

In [163]:
```
f(3) # 13 returned
```

Out [163]: 13

In [164]:
```
function g(x::Int, y::Int) # type restriction
return y, x # explicit return of a tuple
end
g(x::Int, y::Bool) = x * y # add multiple dispatch
g(2, true) # second definition is invoked
```

Out [164]: 2

In [165]:
```
methods(g) # list all methods defined for g
```

Out [165]: 2 methods for generic function **g**:

- g(x::**Int64**, y::**Bool**) at In[164]:4
- g(x::**Int64**, y::**Int64**) at In[164]:2

In [166]:
```
(x -> x^2)(3) # anonymous function with a call
```
Out [166]: 9

In [167]:
```
() -> 0 # anonymous function with no arguments
```
Out [167]: (::#13) (generic function with 1 method)

In [168]:
```
h(x...) = sum(x)/length(x) - mean(x) # vararg function; x is a tuple
h(1, 2, 3) # result is 0
```
Out [168]: 0.0

In [169]:
```
x = (2, 3) # tuple
f(x) # error
```
```
MethodError: no method matching +(::Tuple{Int64,Int64}, ::Int64)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:424
  +(::Complex{Bool}, ::Real) at complex.jl:239
  +(::Char, ::Integer) at char.jl:40
  ...

Stacktrace:
 [1] f(::Tuple{Int64,Int64}) at .\In[162]:1
```

In [170]:
```
f(x...) # OK - tuple unpacking
```
Out [170]: 5

In [171]:
```
s(x; a = 1, b = 1) = x * a / b # function with keyword arguments a and b
s(3, b = 2) # call with keyword argument
```
Out [171]: 1.5

In [172]:
```
x1(; x::Int64 = 2) = x # single keyword argument
x1() # 2 returned
```
Out [172]: 2

In [173]:
```
x2(; x::Bool = true) = x # no multiple dispatch for keyword arguments; function overwritten
x2() # true; old function was overwritten
```
Out [173]: true

In [174]:
```
q(f::Function, x) = 2 * f(x) # simple function wrapper
q(x -> 2x, 10) # 40 returned, no need to use * in 2x (means 2*x)
```
Out [174]: 40

In [175]:
```
q(10) do x # creation of anonymous function by do construct, useful eg. in IO
    2 * x
end
```
Out [175]: 40

In [176]:
```
m = reshape(1:12, 3, 4)
map(x -> x ^ 2, m) # 3x4 array returned with transformed data
```
Out [176]:
```
3×4 Array{Int64,2}:
 1  16  49  100
 4  25  64  121
 9  36  81  144
```

In [177]:
```
filter(x -> bits(x)[end] == '0', 1:12) # a fancy way to choose even integers from the range
```
Out [177]:
```
6-element Array{Int64,1}:
  2
  4
  6
  8
 10
 12
```

As a convention functions with name ending with ! change their arguments in-place. See for example `resize!` in this document.

Default function argument beasts:

In [178]:
```
y = 10
f1(x=y) = x; f1() # 10
```

Out [178]: 10

In [179]:
```
f2(x=y,y=1) = x; f2() # 10
```

Out [179]: 10

In [180]:
```
f3(y=1,x=y) = x; f3() # 1
```

Out [180]: 1

In [181]:
```
f4(;x=y) = x; f4() # 10
```

Out [181]: 10

In [182]:
```
f5(;x=y,y=1) = x; f5() # error - y not defined yet :(
```

```
UndefVarError: y not defined

Stacktrace:
 [1] f5() at .\In[182]:1
```

In [183]:
```
f6(;y=1,x=y) = x; f6() # 1
```

Out [183]: 1

## Variable scoping

The following constructs introduce new variable scope: `function`, `while`, `for`, `try/catch`, `let`, `type`.

You can define variables as:

- `global`: use variable from global scope;
- `local`: define new variable in current scope;
- `const`: ensure variable type is constant (global only).

Special cases:

In [184]:
```
w # error, variable does not exist
```

```
UndefVarError: w not defined
```

In [185]:
```
f() = global w = 1
f() # after the call w is defined globally
w
```

Out [185]: 1

In [186]:
```
function fn(n)
    x = 0
    for i = 1:n
        x = i
    end
    x
end
fn(10) # 10; inside loop we use outer local variable
```

Out [186]: 10

In [187]:
```julia
function fn2(n)
    x = 0
    for i = 1:n
        local x
        x = i
    end
    x
end
fn2(10) # 0; inside loop we use new local variable
```

Out [187]: 0

In [188]:
```julia
function fn3(n)
    for i = 1:n
        local x # this local can be omitted; for introduces new scope
        x = i
    end
    x
end
fn3(10) # x fetched from global scope as it wase already defined
```

Out [188]: (2, 3)

In [189]:
```julia
const n = 2
n = 3 # warning, value changed
```

```
WARNING: redefining constant n
```

Out [189]: 3

In [190]:
```julia
n = 3.0 # error, wrong type
```

```
invalid redefinition of constant n
```

In [191]:
```julia
function fun() # no warning
    const x = 2
    x = true
end
fun() # true, no warning
```

Out [191]: true

Global constants speed up execution.

The `let` rebinds the variable:

In [192]:
```julia
Fs = Array{Any}(2)
i = 1
while i <= 2
    j = i
    Fs[i] = () -> j
    i += 1
end
Fs[1](), Fs[2]() # (2, 2); the same binding for j
```

Out [192]: (2, 2)

In [193]:
```julia
Fs = Array{Any}(2)
i = 1
while i <= 2
    let j = i
        Fs[i] = () -> j
    end
    i += 1
end
Fs[1](), Fs[2]() # (1, 2); new binding for j
```

Out [193]: (1, 2)

In [194]:
```
Fs = Array{Any}(2)
i = 1
for i in 1:2
    j = i
    Fs[i] = () -> j
end
Fs[1](), Fs[2]() # (1, 2); for loops and comprehensions rebind variables
```

Out [194]:
```
(1, 2)
```

## Modules

Modules encapsulate code. Can be reloaded, which is useful to redefine functions and types, as top level functions and types are defined as constants.

In [195]:
```
module M # module name; can be replaced in one session
export xx # what module exposes for the world
xx = 1
y = 2 # hidden variable
end

whos(M) # list exported variables
```

```
                        M      996 bytes  Module
                       xx        8 bytes  Int64
```

In [196]:
```
xx # not found in global scope
```

```
UndefVarError: xx not defined
```

In [197]:
```
M.y # direct variable access possible
```

Out [197]:
```
2
```

In [198]:
```
# import all exported variables
# load standard packages this way
using M

#import variable y to global scope (even if not exported)
import M.y
```

```
WARNING: import of M.y into Main conflicts with an existing identifier; ignored.
```

## Operators

Julia follows standard operators with the following quirks:

In [199]:
```
true || false # binary or operator (singeltons only), || and && use short-circut evaluation
```

Out [199]:
```
true
```

In [ ]:
```
[1 2] .& [2 1] # bitwise and operator
```

In [201]:
```
1 < 2 < 3 # chaining conditions is OK (singeltons only)
```

Out [201]:
```
true
```

In [202]:
```
[1 2] .< [2 1] # for vectorized operators need to add '.' in front
```

Out [202]:
```
1×2 BitArray{2}:
 true  false
```

In [203]:
```
x = [1 2 3]
2x + 2(x+1) # multiplication can be omitted between a literal and a variable or a left parenthesis
```

Out [203]:
```
1×3 Array{Int64,2}:
 6  10  14
```

In [204]:
```
y = [1, 2, 3]
```

Out [204]:
```
3-element Array{Int64,1}:
 1
 2
 3
```

In [205]:
```
x + y # error
```

```
DimensionMismatch("dimensions must match")

Stacktrace:
 [1] promote_shape(::Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}, ::Tuple{Base.OneTo{Int64}}) at .\ir
 [2] +(::Array{Int64,2}, ::Array{Int64,1}) at .\arraymath.jl:37
```

In [206]:
```
x .+ y # 3x3 matrix, dimension broadcasting
```

Out [206]:
```
3×3 Array{Int64,2}:
 2  3  4
 3  4  5
 4  5  6
```

In [207]:
```
x + y' # 1x3 matrix
```

Out [207]:
```
1×3 Array{Int64,2}:
 2  4  6
```

In [208]:
```
x * y # array multiplication, 1-element vector (not scalar)
```

Out [208]:
```
1-element Array{Int64,1}:
 14
```

In [209]:
```
x .* y # element-wise multiplication, 3x3 array
```

Out [209]:
```
3×3 Array{Int64,2}:
 1  2  3
 2  4  6
 3  6  9
```

In [210]:
```
x == [1 2 3] # true, object looks the same
```

Out [210]:
```
true
```

In [211]:
```
x === [1 2 3] # false, objects not identical
```

Out [211]:
```
false
```

In [212]:
```
z = reshape(1:9, 3, 3)
```

Out [212]:
```
3×3 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
 1  4  7
 2  5  8
 3  6  9
```

In [213]:
```
z + x # error
```

```
DimensionMismatch("dimensions must match")

Stacktrace:
 [1] promote_shape(::Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}, ::Tuple{Base.OneTo{Int64},Base.OneT
 [2] promote_shape(::Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}, ::Array{Int64,2}) at .\ir
 [3] +(::Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}, ::Array{Int64,2}) at .\arraymath.jl:3
```

In [214]:
```
z .+ x # x broadcasted vertically
```

Out [214]:
```
3×3 Array{Int64,2}:
 2  6  10
 3  7  11
 4  8  12
```

In [215]:
```
z .+ y # y broadcasted horizontally
```

Out [215]:
```
3×3 Array{Int64,2}:
 2  5   8
 4  7  10
```

```
 6  9  12
```

In [216]:
```
# explicit broadcast of singelton dimensions
# function + is called for each array element
broadcast(+, [1 2], [1; 2])
```

Out [216]:
```
2×2 Array{Int64,2}:
 2  3
 3  4
```

Many typical matrix transformation functions are available (see documentation).

## Essential general usage functions

In [217]:
```
show(collect(1:100)) # show text representation of an object
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
```

In [218]:
```
eps() # distance from 1.0 to next representable Float64
```

Out [218]:
```
2.220446049250313e-16
```

In [219]:
```
nextfloat(2.0) # next float representable, similarly provided prevfloat
```

Out [219]:
```
2.0000000000000004
```

In [220]:
```
isequal(NaN, NaN) # true
```

Out [220]:
```
true
```

In [221]:
```
NaN == NaN # false
```

Out [221]:
```
false
```

In [222]:
```
NaN === NaN # true
```

Out [222]:
```
true
```

In [223]:
```
isequal(1, 1.0) # true
```

Out [223]:
```
true
```

In [224]:
```
1 == 1.0 # true
```

Out [224]:
```
true
```

In [225]:
```
1 === 1.0 # false
```

Out [225]:
```
false
```

In [226]:
```
isfinite(Inf) # false, similarly provided: isinf, isnan
```

Out [226]:
```
false
```

In [227]:
```
fld(-5, 3), mod(-5, 3) # (-2, 1), division towards minus infinity
```

Out [227]:
```
(-2, 1)
```

In [228]:
```
div(-5, 3), rem(-5, 3) # (-1, -2), division towards zero
```

Out [228]:
```
(-1, -2)
```

In [229]:
```
find(x -> mod(x, 2) == 0, 1:8) # find indices for which function returns true
```

Out [229]:
```
4-element Array{Int64,1}:
 2
 4
 6
 8
```

In [230]:
```
x = [1 2]; identity(x) === x # true, identity function
```
Out [230]:
```
true
```
In [231]:
```
info("Info") # print information, similarly warn and error (raises error)
```
```
□[1m□[36mINFO: □[39m□[22m□[36mInfo
□[39m
```
In [232]:
```
ntuple(x->2x, 3) # create tuple by calling x->2x with values 1, 2 and 3
```
Out [232]:
```
(2, 4, 6)
```
In [233]:
```
isdefined(:x) # if variable x is defined (:x is a symbol)
```
Out [233]:
```
true
```
In [234]:
```
y = Array{Any}(2); isassigned(y, 3) # if position 3 in array is assigned (not out of bounds or #un
```
Out [234]:
```
false
```
In [235]:
```
fieldtype(typeof(1:2),:start) # get type of the field in composite type (passed as symbol)
```
Out [235]:
```
Int64
```
In [236]:
```
fieldnames(typeof(1:2))
```
Out [236]:
```
2-element Array{Symbol,1}:
 :start
 :stop
```
In [237]:
```
1:5 |> exp |> sum # function application chaining
```
```
□[1m□[33mWARNING: □[39m□[22m□[33mexp{T <: Number}(x::AbstractArray{T}) is deprecated, use exp.(x)
instead.□[39m
Stacktrace:
 [1] □[1mdepwarn□[22m□[22m□[1m(□[22m□[22m::String, ::Symbol□[1m)□[22m□[22m at
□[1m.\deprecated.jl:64□[22m□[22m
 [2] □[1mexp□[22m□[22m□[1m(□[22m□[22m::UnitRange{Int64}□[1m)□[22m□[22m at
□[1m.\deprecated.jl:51□[22m□[22m
 [3] □[1m|>□[22m□[22m□[1m(□[22m□[22m::UnitRange{Int64}, ::Base.#exp□[1m)□[22m□[22m at
□[1m.\operators.jl:857□[22m□[22m
 [4] □[1minclude_string□[22m□[22m□[1m(□[22m□[22m::String, ::String□[1m)□[22m□[22m at
□[1m.\loading.jl:498□[22m□[22m
 [5] □[1mexecute_request□[22m□[22m□[1m(□[22m□[22m::LastMain.ZMQ.Socket,
::LastMain.IJulia.Msg□[1m)□[22m□[22m at
□[1mD:\Software\JULIA_PKG\v0.6\IJulia\src\execute_request.jl:156□[22m□[22m
 [6] □[1meventloop□[22m□[22m□[1m(□[22m□[22m::LastMain.ZMQ.Socket□[1m)□[22m□[22m at
□[1mD:\Software\JULIA_PKG\v0.6\IJulia\src\eventloop.jl:8□[22m□[22m
 [7] □[1m(::LastMain.IJulia.##9#12)□[22m□[22m□[1m(□[22m□[22m□[1m)□[22m□[22m at
□[1m.\task.jl:335□[22m□[22m
while loading In[237], in expression starting on line 1
```
Out [237]:
```
233.2041839862982
```
In [238]:
```
zip(1:3, 1:3) |> collect # convert iterables to iterable tuple and pass it to collect
```
Out [238]:
```
3-element Array{Tuple{Int64,Int64},1}:
 (1, 1)
 (2, 2)
 (3, 3)
```
In [239]:
```
enumerate("abc") # create iterator of tuples (index, collection element)
```
Out [239]:
```
Enumerate{String}("abc")
```
In [240]:
```
  collect(enumerate("abc"))
```
Out [240]:
```
3-element Array{Tuple{Int64,Char},1}:
 (1, 'a')
 (2, 'b')
 (3, 'c')
```
In [241]:
```
isempty("abc") # check if collection is empty
```

Out [241]: false

In [242]:
```
'b' in "abc" # check if element is in a collection
```

Out [242]: true

In [243]:
```
indexin(collect("abc"), collect("abrakadabra")) # [11, 9, 0] ('c' not found), needs arrays
```

Out [243]:
```
3-element Array{Int64,1}:
 11
  9
  0
```

In [244]:
```
findin("abc", "abrakadabra") # [1, 2] ('c' was not found)
```

Out [244]:
```
2-element Array{Int64,1}:
 1
 2
```

In [245]:
```
unique("abrakadabra") # return unique elements
```

Out [245]:
```
5-element Array{Char,1}:
 'a'
 'b'
 'r'
 'k'
 'd'
```

In [246]:
```
issubset("abc", "abcd") # check if every element in fist collection is in the second
```

Out [246]: true

In [247]:
```
indmax("abrakadabra") # index of maximal element (3 - 'r' in this case)
```

Out [247]: 3

In [248]:
```
findmax("abrakadabra") # tuple: maximal element and its index
```

Out [248]: ('r', 3)

In [249]:
```
filter(x->mod(x,2)==0, 1:10) # retain elements of collection that meet predicate
```

Out [249]:
```
5-element Array{Int64,1}:
  2
  4
  6
  8
 10
```

In [250]:
```
dump(1:2:5) # show all user-visible structure of an object
```

```
StepRange{Int64,Int64}
  start: Int64 1
  step: Int64 2
  stop: Int64 5
```

In [251]:
```
sort(rand(10)) # sort 10 uniform random variables
```

Out [251]:
```
10-element Array{Float64,1}:
 0.0146516
 0.305749
 0.408407
 0.524122
 0.608936
 0.685384
 0.777021
 0.826193
 0.855542
 0.993751
```

## Reading and writing data

For I/O details refer documentation. Basic operations:

- `readdlm`, `readcsv`: read from file
- `writedlm`, `writecsv`: write to a file

Warning! Trailing spaces are not discarded if `delim=' '` in file reading.

## Random numbers

Basic random numbers:

In [252]:
```
srand(1) # set random number generator seed to 1
rand() # generate random number from U[0,1)
```

Out [252]:
```
0.23603334566204692
```

In [253]:
```
rand(3, 4) # generate 3x4 matrix of random numbers from U[0,1]
```

Out [253]:
```
3×4 Array{Float64,2}:
 0.346517    0.488613  0.999905  0.555751
 0.312707    0.210968  0.251662  0.437108
 0.00790928  0.951916  0.986666  0.424718
```

In [254]:
```
rand(2:5, 10) # generate vector of 10 random integer numbers in range form 2 to 5
```

Out [254]:
```
10-element Array{Int64,1}:
 4
 2
 3
 4
 5
 5
 3
 4
 3
 5
```

In [255]:
```
randn(10) # generate vector of 10 random numbers from standard normal distribution
```

Out [255]:
```
10-element Array{Float64,1}:
  0.795949
  0.670062
  0.550852
 -0.0633746
  1.33694
 -0.0731486
 -0.745464
 -1.22006
 -0.0531773
 -0.165136
```

Advanced randomness form `Distributions` package:

In [ ]:
```
using Distributions # load package
sample(1:10, 10) # single bootstrap sample from set 1-10
```

```
[1m[36mINFO: [39m[22m[36mRecompiling stale cache file
D:\Software\JULIA_PKG\lib\v0.6\Distributions.ji for module Distributions.
[39m
```

In [ ]:
```
b = Beta(0.4, 0.8) # Beta distribution with parameters 0.4 and 0.8
# see documentation for supported distributions
```

In [ ]:
```
mean(b) # expected value of distribution b
# see documentation for other supported statistics
```

In [ ]:
```
rand(b, 100) # 100 independent random samples from distribution b
```

## Statistics and machine learning

Visit http://juliastats.github.io/ for the details (in particular R-like data frames).

Starting with Julia version 0.4 there is a core language construct `Nullable` that allows to represent missing value (similar to HaskellMaybe).

In [ ]:
```
u1 = Nullable(1) # contains value
u2 = Nullable{Int64}() # missing value
get(u1) # OK
```

In [ ]:
```
get(u2) # error - missing
```

In [ ]:
```
isnull(u1) # false
```

In [ ]:
```
isnull(u2) # true
```

## Plotting

There are several plotting packages for Julia: `PyPlot`, `Gadfly`, `Plots`, ....

In [ ]:
```
using PyPlot
srand(1) # second plot
x, y = randn(100), randn(100)
plot(x, y)
```

## Macros

You can define macros (see documentation for details). Useful standard macros.

Assertions:

In [ ]:
```
@assert 1 == 2 "ERROR" # 2 macro arguments; error raised
```

In [ ]:
```
using Base.Test # load Base.Test module
@test 1 == 2 # similar to assert; error
```

In [ ]:
```
@test_approx_eq 1 1.1 # error
```

In [ ]:
```
@test_approx_eq_eps 1 1.1 0.2 # no error
```

Function vectorization:

In [ ]:
```
t(x::Float64, y::Float64 = 1.0) = x * y
```

In [ ]:
```
t(1.0, 2.0) # OK
```

In [ ]:
```
t([1.0 2.0]) # error
```

In [ ]:
```
t.([1.0 2.0]) # OK
```

In [ ]:
```
t([1.0 2.0], 2.0) # error
```

In [ ]:
```
t.([1.0 2.0], 2.0) # OK
```

In [ ]:
```
t.(2.0, [1.0 2.0]) # OK
```

Benchmarking:

In [ ]:
```
@time [x for x in 1:10^6].' # print time and memory
```

In [ ]:
```
@timed [x for x in 1:10^6].' # return value, time and memory
```

In [ ]:
```
@elapsed [x for x in 1:10^6] # return time
```

In [ ]:
```
@allocated [x for x in 1:10^6] # return memory
```

In [ ]:
```
tic() # start timer
```

In [ ]:
```
toc() # stop timer and print time
```

In [ ]:
```
tic()
toq()
```