The Julia Express

Bogumił Kamiński

August 30, 2018

Contents

1 Introduction

The Purpose of this document is to introduce programmers to Julia programming by example. This is a simplified exposition of the language.[1]

It is best to execute these examples by copy-pasting to Julia REPL or copying them to a file and next running them using `include` function. The difference is that copy-paste approach will echo output of each instruction to the terminal.

If some package is missing on your system switch to package manager mode by pressing `]` in Julia REPL, and then write `add [package name]` to require installing it.[2]

Major stuff not covered (please see the documentation):

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

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

Julia Express was tested using the following 64-bit Julia version:

```
versioninfo()
# Julia Version 1.0.0
# Commit 5d4eaca0c9 (2018-08-08 20:58 UTC)
# Platform Info:
#   OS: Windows (x86_64-w64-mingw32)
#   CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
#   WORD_SIZE: 64
#   LIBM: libopenlibm
#   LLVM: libLLVM-6.0.0 (ORCJIT, broadwell)
```

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

2 Getting around

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

1) `^D` (exits Julia);
2) `^C` (interrupts computations);
3) `?` (enters help mode);
4) `;` (enters system shell mode);
5) `]` (enters package manager mode);
6) putting `;` after the expression will disable showing its value in REPL (not needed in scripts).

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

```
@edit max(1,2)      # show the definition of max function when invoked with arguments 1 and 2
varinfo()           # list of global variables and their types
cd("D:/")           # change working directory to D:/ (on Windows)
pwd()               # get current working directory
include("file.jl")  # execute source file
exit(1)             # exit Julia with code 1 (exit code 0 by default)
clipboard([1,2])    # copy data to system clipboard
clipboard()         # load data from system clipboard as string
```

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):

```
"""Sieve of Eratosthenes function docstring"""
```

[1] The rocket ship clip is free for download at http://www.clipartlord.com/free-cartoon-rocketship-clip-art-2/.

[2] Sometimes a package might fail to load using `using [package name]`. The reason might be that Julia 1.0 was just released and not all packages have tagged their latest versions. If you have this problem then installing the package using `add [package name]#master` on package that throws an error should solve the problem. All examples given in this documented were tested to work under Julia 1.0.

```
function es(n::Int) # accepts one integer argument
    isprime = trues(n) # n-element vector of true-s
    isprime[1] = false # 1 is not a prime
    for i in 2:isqrt(n) # loop odd integers less or equal than sqrt(n)
        if isprime[i] # conditional evaluation
            for j in i^2: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(10^6))   # check function execution time and memory usage
```

3 Basic literals and types

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

```
1::Int                # 64 bit integer on 64 bit machine, no overflow warnings
1.0::Float64          # 64 bit float, defines NaN, -Inf, Inf
true::Bool            # boolean, allows "true" and "false"
'c'::Char             # character, allows Unicode
"s"::AbstractString   # strings, allows Unicode, see also Strings below
```

All basic types above are immutable. Specifying type assertion is optional (and usually it is not needed, but I give it to show how you the names of types). Type assertions for variables are made in the same way and they can be useful to catch bugs in your code.

An important feature of integers in Julia is that by default they are 64 bit on 64 bit machine and 32 bit on 32 bit machine. This means that `1::Int32` assertion will fail on 64-bit version. Notably `Int` is a constant whose value is either `Int64` or `Int32` depending on version (the same with unsigned integer `UInt`).

There is no automatic type conversion, unless some function explicitly performs it. This is especially important in function calls. Examples of conversions:

```
Int64('a')     # character to integer
Int64(2.0)     # float to integer
Int64(1.3)     # inexact error
Int64("a")     # error no conversion possible
Float64(1)     # integer to float
Bool(1)        # converts to boolean true
Bool(0)        # converts to boolean false
Bool(2)        # conversion error
Char(89)       # integer to char
string(true)   # cast Bool to string (works with other types, note small caps)
string(1,true) # string can take more than one argument and concatenate them
zero(10.0)     # zero of type of 10.0
one(Int64)     # one of type Int64
```

General conversion can be done using `convert(Type, x)` (typically `convert` would fall back to a type constructor):

```
convert(Int64, 1.0) # convert float to integer
```

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

```
parse(Int64, "1") # parse "1" string as Int64
```

Automatic promotion of many arguments to common type (if any) can be achieved using `promote`:

```
promote(true, BigInt(1)//3, 1.0) # tuple (see Tuples) of BigFloats, true promoted to 1.0
promote("a", 1)                  # promotion to common type not possible
```

Many operations (arithmetic, assignment) are defined in a way that performs automatic type promotion (so this is a way to work around no automatic type conversion rule in Julia).

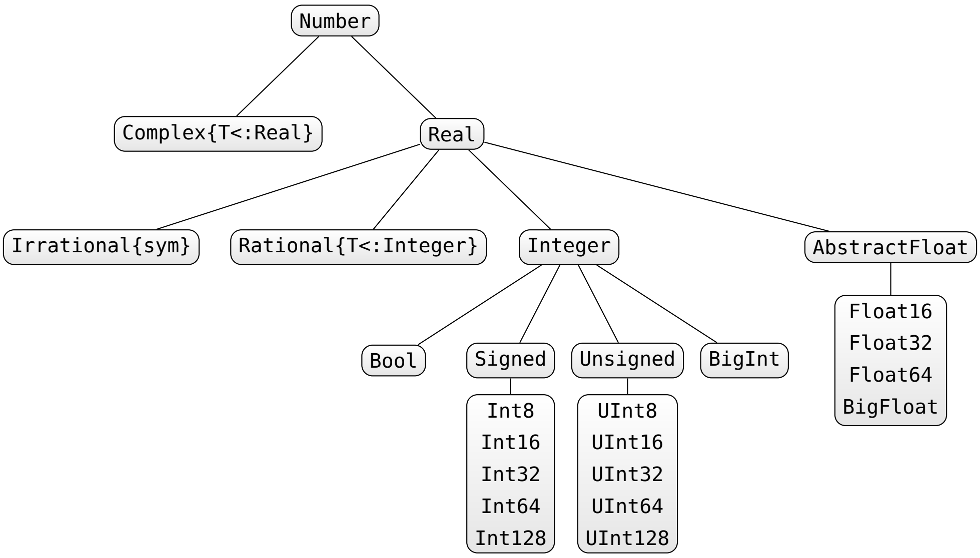

Figure 1: Hierarchy of numeric types

One can verify type of a value in the following way:

```
typeof("abc")              # String returned which is a AbstractString subtype
isa("abc", AbstractString) # true
isa(1, Float64)            # false, integer is not a float
isa(1.0, Float64)          # true
isa(1.0, Number)           # true, Number is abstract type
supertype(Int64)           # supertype of Int64
subtypes(Real)             # subtypes of abstract type Real
```

It is possible to perform calculations using arbitrary precision arithmetic, complex and rational numbers:

```
BigInt(10)^1000   # big integer
BigFloat(10)^1000 # big float, see documentation how to change default precision
1 + 1im           # complex number
123//456          # rational numbers are created using // operator
```

Type hierarchy of all standard numeric types is given in Figure 1.

4 Special literals and types

Type beasts:

```
Any     # all objects are of this type
Union{} # subtype of all types, no object can have this type
Nothing # type indicating nothing, subtype of Any
nothing # only instance of Nothing
Missing # type indicating missing value
missing # only instance of Missing
```

Additionally `#undef` indicates an incompletely initialized object element (see documentation for details).

4.1 Tuples and NamedTuples

Tuples are immutable sequences indexed from 1:

```
()            # empty tuple
(1,)          # one element tuple
("a", 1)      # two element tuple
('a', false)::Tuple{Char, Bool} # tuple type assertion
x = (1, 2, 3)
x[1]          # 1 (element)
```

```
x[1:2]        # (1, 2) (tuple)
x[4]          # bounds error
x[1] = 1      # error - tuple is not mutable
a, b = x      # tuple unpacking a==1, b==2
```

Additionally you can add names to tuple entries (via named tuples):

```
NamedTuple()  # empty named tuple
(a=1,)        # one element named tuple
(x="a", y=1)  # two element named tuple
x = (p=1, q=2, r=3)
x.p           # access to element p of a tuple
typeof(x)     # NamedTuple{(:p, :q, :r),Tuple{Int64,Int64,Int64}}, field names are part of type
```

4.2 Arrays

Arrays are mutable and passed by reference. Array creation:

```
Array{Char}(undef, 2, 3, 4) # uninitialized 2x3x4 array of Chars
Array{Int64}(undef, 0, 0)   # degenerate 0x0 array of Int64
zeros(5)             # vector of Float64 zeros
ones(5)              # vector of Float64 ones
ones(Int64, 2, 1)    # 2x1 array of Int64 ones
trues(3), falses(3)  # tuple of vector of trues and of falses
Matrix(I, 3, 3)      # 3x3 Bool identity matrix, requires to run first: using LinearAlgebra
x = range(0, stop=1, length=11) # iterator having 11 equally spaced elements
collect(x)           # converts iterator to vector
1:10                 # iterable from 1 to 10
1:2:10               # iterable from 1 to 9 with 2 skip
reshape(1:12, 3, 4)  # 3x4 array filled with 1:12 values
fill("a", 2, 2)      # 2x2 array filled with "a"
repeat(rand(2,2), 3, 2) # 2x2 random matrix repeated 3x2 times
x = [1, 2]           # two element vector
resize!(x, 5)        # resize x in place to hold 5 values (filled with garbage)
[1]                  # vector with one element (not a scalar)
[x * y for x in 1:2, y in 1:3] # comprehension generating 2x3 array
Float64[x^2 for x in 1:4] # casting comprehension result element type to Float64
[1 2]                # 1x2 matrix (hcat function)
[1 2]'               # 2x1 Adjoint matrix (reuses memory)
permutedims([1 2])   # 2x1 matrix (permuted dimensions, new memory)
[1, 2]               # vector (concatenation)
[1; 2]               # vector (vcat function)
[1 2 3; 1 2 3]       # 2x3 matrix (hvcat function)
[1; 2] == [1 2]'     # false, different array dimensions
[(1, 2)]             # 1-element vector
collect((1, 2))      # 2-element vector by tuple unpacking
[[1 2] 3]            # concatenate along rows (hcat)
[[1; 2]; 3]          # concatenate along columns (vcat)
```

Vectors (1D arrays) are treated as column vectors.

Most of the functionality for working with matrices are in `LinearAlgebra` package. Additionally Julia offers sparse and distributed matrices (see documentation for details).

Commonly needed array utility functions:

```
a = [x * y for x in 1:2, y in 1, z in 1:3] # 2x3 array of Int64; singleton dimension is dropped
a = [x * y for x in 1:2, y in 1:1, z in 1:3] # 2x1x3 array of Int64; singleton dimension is not dropped
ndims(a)             # number of dimensions in a
eltype(a)            # type of elements in a
length(a)            # number of elements in a
size(a)              # tuple containing dimension sizes of a
axes(a)              # tuple of ranges specifying array axes
eeachindex(a)        # each index to an array a
```

```
vec(a)                 # cast array to vector (single dimension)
squeeze(a, dims=2)     # remove 2nd dimension as it has size 1
sum(a, dims=3)         # calculate sums for 3rd dimensions, similarly: mean, std,
                       # prod, minimum, maximum, any, all
count(x -> x > 0, a) # count number of times a predicate is true, similar: all, any
```

Access functions:

```
a = 0:0.01:1           # range with step 0.01
a[1]                   # get scalar 0.0
a[end]                 # get scalar 1.0 (last position)
a[1:2:end]             # every second element from range
view(a, 1:2:101)       # a view into a subarray of a
a[[1, 3, 6]]           # 1st, 3rd and 6th element of a, Array{Float64,1}
lastindex(a)           # last index of the collection a; similarly firstindex
```

Observe the treatment of trailing singleton dimensions:

```
a = reshape(1:12, 3, 4)
a[:, 1:2]            # 3x2 matrix
a[:, 1]              # 3 element vector
a[1, :]              # 4 element vector
a[1:1, :]            # 1x4 matrix
a[:, :, 1, 1]        # works 3x4 matrix
a[:, :, :, [true]] # wroks 3x4x1x1 matrix
a[1, 1, [false]]     # works 0-element Array{Int64,1}
```

Array assignment:

```
x = collect(reshape(1:8, 2, 4))
x[:,2:3] = [1 2]             # error; size mismatch
x[:,2:3] .= [1 2]            # OK, broadcasting with .
x[:,2:3] = repeat([1 2], 2) # OK
x[:,2:3] .= 3                # OK, need to use broadcast with .
```

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

```
x = Array{Any}(undef, 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
a               # identical as x
b               # only x[2][1] changed from original x
c               # contents of original x
```

Array types syntax examples:

```
[1 2]::Array{Int64, 2}      # 2 dimensional array of Int64
[true; false]::Vector{Bool} # vector of Bool
[1 2; 3 4]::Matrix{Int64}   # matrix of Int64
```

4.3 Composite types

You can define and access composite types:

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

```
p.x               # access field
p.meta = 2        # change field value
p.x = 1.5         # error, wrong data type
p.z = 1           # error - no such field
fieldnames(Point) # get names of type fields
```

You can define type to be immutable by removing `mutable`.

There are also union types (see documentation of *Type Unions* in the Julia manual for details).

Finally you can define that your type is a subtype of an abstract type (see documentation of *Abstract Types* in the Julia manual for details).

4.4 Dictionaries

Associative collections (key-value dictionaries):

```
x = Dict{Float64, Int64}()       # empty dictionary mapping floats to integers
y = Dict("a"=>1, "b"=>2)         # filled dictionary
y["a"]                           # element retrieval
y["c"]                           # error
y["c"] = 3                       # added element
haskey(y, "b")                   # check if y contains key "b"
keys(y), values(y)               # tuple of collections returning keys and values in y
delete!(y, "b")                  # delete key from a collection, see also: pop!
get(y,"c","default")             # return y["c"] or "default" if not haskey(y,"c")
```

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

5 Strings

String operations:

```
"Hi " * "there!"       # string concatenation
"Ho " ^ 3              # repeat string
string("a = ", 123.3)  # create using print function
repr(123.3)            # fetch value of show function to a string
occursin("CD", "ABCD") # check if first string contains second
"\"\n\t\$"             # C-like escaping in strings, new \$ escape
x = 123
"$x + 3 = $(x+3)"      # unescaped $ is used for interpolation
"\$199"                # to get a $ symbol you must escape it
```

PCRE regular expressions handling:

```
r = r"A|B"           # create new regexp
occursin(r, "CD")    # false, no match found
m = match(r, "ACBD") # find first regexp match, see documentation for details
```

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

6 Programming constructs

The simplest way to create new variable is by assignment:

```
x = 1.0          # x is bound to Float64 value
x = 1            # now x is bound to value Int32 on 32 bit machine and Int64 on 64 bit machine
```

Expressions can be compound using `;` or `begin end` block:

```
x = (a = 1; 2 * a) # after: x = 2; a = 1
y = begin
  b = 3
  3 * b
end              # after: y = 9; b = 3
```

There are standard programming constructs:

```
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

1==2 ? "A" : "B" # standard ternary operator

i = 1
while true
    global i += 1 # global would not be needed if the loop were inside a function
    if i > 10
      break
    end
end

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 defined in the inner scope of the loop
```

You can define your own functions:

```
f(x, y = 10) = x + y           # one line definition of a new function f with y defaulting to 10
                               # last expression result returned
function f(x, y=10)            # the same as above but allowing multiple expressions
    x + y                      # in the body of the function
end
f(3, 2)                        # simple call, 5 returned
f(3)                           # 13 returned
(x -> x^2)(3)                  # anonymous function with a call example
() -> 0                        # anonymous function with no arguments
h(x...) = sum(x)/length(x) - mean(x) # vararg function; x is a tuple
h(1, 2, 3)                     # result is 0
x = (2, 3)                     # tuple
f(x)                           # error
f(x...)                        # OK - tuple unpacking
s(x; a = 1, b = 1) = x * a / b # function with keyword arguments a and b
s(3, b = 2)                    # call with keyword argument
q(f::Function, x) = 2 * f(x)   # function can be passed around
q(x -> 2x, 10)                 # 40 returned, no need to use * in 2x (means 2*x)
q(10) do x                     # creation of anonymous function by do construct, useful eg. in IO
  2 * x
end
m = reshape(1:12, 3, 4)
map(x -> x ^ 2, m)             # 3x4 array returned with transformed data
filter(x -> bitstring(x)[end] == '0', 1:12)  # a fancy way to choose even integers from the range
==(1)                          # returns a function that test for equality
findall(==(1), 1:10)           # find indices of all elements equal to 1, similar: findfirst, findlast
```

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

Default function arguments are evaluated left to right:

```
y = 10
f1(x=y) = x; f1()      # 10
f2(x=y,y=1) = x; f2()  # 10
```

```
f3(y=1,x=y) = x; f3()  # 1
f4(;x=y) = x; f4()     # 10
f5(;x=y,y=1) = x; f5() # 10
f6(;y=1,x=y) = x; f6() # 1
```

There is an important part of Julia terminology is that a *function* can have multiple *methods*. Each method specifies a behavior of a function for a given set of argument types. This behavior is called multiple dispatch and works only for positional arguments. Here are some short examples. More details are given in *Methods* section of the Julia manual.

```
g(x, y) = println("all accepted") # method for g function accepting any type of x and y
function g(x::Int, y::Int)        # method called when both x and y are Int
  y, x
end
g(x::Int, y::Bool) = x * y        # this will be called when x is Int and y is Bool
g(1.0, 1)                         # first definition is invoked
g(1, 1)                           # second definition is invoked
g(1, true)                        # third definition is invoked
methods(g)                        # list all methods defined for g
t(; x::Int64 = 2) = x             # a single keyword argument
t()                               # 2 returned
t(; x::Bool = true) = x           # no multiple dispatch for keyword arguments; function overwritten
t()                               # true; old function was overwritten
```

7 Variable scoping

The following constructs introduce new variable scope: `function`, `while`, `for`, `try/catch`, `let`, `struct`, `mutable struct`. You can define variables as:

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

Special cases:

```
t                  # error, variable does not exist
f() = global t = 1
f()                # after the call t is defined globally

function f1(n)
  x = 0
  for i = 1:n
    x = i
  end
  x
end
f1(10)            # 10; inside loop we use outer local variable

function f2(n)
  x = 0
  for i = 1:n
    local x
    x = i
  end
  x
end
f2(10)            # 0; inside loop we use new local variable

function f3(n)
  for i = 1:n
    x = i
```

```
  end
  x
end
f3(10)            # error; x not defined in outer scope

const x = 2
x = 3 # warning, value changed
x = 3.0 # error, wrong type
```

Global constants speed up execution.

Loops and comprehensions rebind variables on each iteration, so they are safe to use then creating closures in iteration:

```
Fs = Array{Any}(undef, 2)
for i in 1:2
  Fs[i] = () -> i
end
Fs[1](), Fs[2]() # (1, 2)
```

8 Modules

Modules encapsulate code and each module has its own global name space (module name of Julia REPL is `Main`).

```
module M # module name
export x # what module exposes for the world
x = 1
y = 2 # hidden variable
end

varinfo(M) # list exported variables
x       # not found in global scope
M.y     # direct variable access possible

# import all exported variables
# also load standard packages this way, but without . prefix
using .M

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

Rebinding variables defined in other modules is not allowed.

9 Operators

Julia follows standard operators with the following quirks:

```
true || false    # binary or operator (singletons only), || and && use short-circut evaluation
[1 2] .& [2 1]    # bitwise and operator (vectorized by .)
1 < 2 < 3        # chaining conditions is OK (singletons only without .)
[1 2] .< [2 1]   # for vectorized operators need to add '.' in front
x = [1 2 3]
2x + 2(x .+ 1)      # multiplication can be omitted between a literal and a variable or a left parenthesis
y = [1, 2, 3]
x + y  # error
x .+ y # 3x3 matrix, dimension broadcasting
x + y' # 1x3 matrix
x * y  # array multiplication, 1-element vector (not scalar)
x .* y # element-wise multiplication, 3x3 array

x == [1 2 3]  # true, object looks the same
x === [1 2 3] # false, objects not identical
```

```
z = reshape(1:9, 3, 3)
z + x  # error
z .+ x # x broadcasted vertically
z .+ y # y broadcasted horizontally

# explicit broadcast of singleton dimensions
# function + is called for each array element
broadcast(+, [1 2], [1; 2])

# broadcasting using . operator
using Random
length([randstring(10) for i in 1:5]) # 5 - length of an array
length.([randstring(10) for i in 1:5]) # 5-element array of 10s - lengths of strings
```

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

10 Essential general usage functions

```
show(collect(1:100)) # show text representation of an object
eps()              # distance from 1.0 to next representable Float64
nextfloat(2.0)     # next float representable, similarly provided prevfloat
isequal(NaN, NaN) # true
NaN == NaN        # false
NaN === NaN       # true
isequal(1, 1.0)   # true
1 == 1.0          # true
1 === 1.0         # false
0.0 == -0.0       # true
0.0 === -0.0      # false
isfinite(Inf)     # false, similarly provided: isinf, isnan
fld(-5, 3), mod(-5, 3) # (-2, 1), division towards minus infinity
div(-5, 3), rem(-5, 3) # (-1, -2), division towards zero
findall(x -> mod(x, 2) == 0, 1:8) # find indices for which function returns true
x = [1 2]; identity(x) === x # true, identity function
@info "Info"      # print information, similarly @warn and @error (see Logging module)
ntuple(x->2x, 3)  # create tuple by calling x->2x with values 1, 2 and 3
@isdefined x      # if variable x is defined
y = Array{Any}(undef,2); isassigned(y, 3)  # if position 3 in array is assigned (not out of bounds or #undef)
fieldtype(typeof(1:2),:start) # get type of the field in composite type (passed as symbol)
fieldnames(typeof(1:2)) # get field names of a type
zip(1:3, 1:3) |> collect # convert iterables to iterable tuple and pass it to collect
enumerate("abc")  # create iterator of tuples (index, collection element)
collect(enumerate("abc"))
isempty("abc")    # check if collection is empty; strings are treated as collections of characters
'b' in "abc"      # check if element is in a collection
indexin(collect("abc"), collect("abrakadabra")) # [1, 2, nothing] ('c' not found), needs arrays
findall(in("abrakadabra"), "abc") # [1, 2] ('c' was not found)
unique("abrakadabra") # return unique elements
issubset("abc", "abcd") # check if every element in fist collection is in the second
argmax("abrakadabra") # index of maximal element (3 - 'r' in this case)
findmax("abrakadabra") # tuple: maximal element and its index
filter(x->mod(x,2)==0, 1:10) # retain elements of collection that meet predicate
dump(1:2:5)       # show all user-visible structure of an object
sort(rand(10))    # sort 10 uniform random values, sort! for in-place operation
```

11 Reading and writing data

For I/O details refer documentation. There are numerous packages providing this functionality. Basic operations from `DelimitedFiles` package:

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

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

12 Random numbers

Basic random numbers:

```
Random.seed!(1) # set random number generator seed to 1; needs calling first: using Random
rand()         # generate random number from U[0,1)
rand(3, 4)     # generate 3x4 matrix of random numbers from U[0,1]
rand(2:5, 10) # generate vector of 10 random integer numbers in range form 2 to 5
randn(10)      # generate vector of 10 random numbers from standard normal distribution
```

Advanced randomness form `Distributions` package TODO: FIXME:

```
using Distributions # load package
sample(1:10, 10)    # single bootstrap sample from set 1-10
b = Beta(0.4, 0.8)  # Beta distribution with parameters 0.4 and 0.8
                    # see documentation for supported distributions
mean(b)             # expected value of distribution b
                    # see documentation for other supported statistics
rand(b, 100)        # 100 independent random samples from distribution b
```

13 Statistics and machine learning

Visit `http://juliastats.github.io/` for the details (in particular R-like data frames).
Starting with Julia version 1.0 there is a core language construct `Missing` that allows to represent missing value.

```
missing # Missing value
ismissing(missing) # true
coalesce(missing, 1, 2) # return first non-missing value, or missing if all are missing
```

14 Macros

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

```
@assert 1 == 2 "ERROR"            # 2 macro arguments; error raised
using Test                        # load Test package
@test 1 == 2                      # similar to assert; error
@test_throws DomainError sqrt(-1) # passed, sqrt(-1) is not possible
```

Function broadcasting:

```
t(x::Float64, y::Float64 = 1.0) = x * y
t(1.0, 2.0)                # OK
t([1.0 2.0])               # error
t.([1.0 2.0])              # OK
t([1.0 2.0], 2.0)          # error
t.([1.0 2.0], 2.0)         # OK
t.(2.0, [1.0 2.0])         # OK
t.([1.0 2.0], [1.0 2.0])   # OK
t.([1.0, 2.0], [1.0 2.0]) # OK
```

Benchmarking:

```
@time [x for x in 1:10^6];    # print time and memory
@timed [x for x in 1:10^6];   # return value, time and memory
@elapsed [x for x in 1:10^6]   # return time
@allocated [x for x in 1:10^6] # return memory
```

Use `BenchmarkTools` package for a more powerful functionality.

15 Plotting

There are several plotting packages for Julia: `Plots`, `Gadfly` and `PyPlot`. Here we show how to use on `PyPlot`.

```
using PyPlot
using Random
Random.seed!(1) # make the plot reproducible
x, y = randn(100), randn(100)
scatter(x, y)
```

A simple example adapting
https://matplotlib.org/1.2.1/examples/pylab_examples/histogram_demo.html:

```
using Distributions
using PyPlot

mu, sigma = 100, 15
x = mu .+ sigma * randn(10000)

n, bins, patches = plt[:hist](x, 50, density=1,
    facecolor="green", alpha=0.75)

y = pdf.(Ref(Normal(mu, sigma)), bins);
plot(bins, y, "r--", linewidth=1)

xlabel("Smarts")
ylabel("Probability")
title(raw"$\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$")
axis([40, 160, 0, 0.03])
grid(true)
```

producing: