Julia Cheatsheet
by Blair Labatt III, page 1 of 2

1 Invocation

Environment Variables

Edit ~.juliarc for persistent Δs. Other env-variables for debugging, REPL color scheme, & parallelization control here.

- JULIA_EDITOR
- JULIA_PROJECT
- JULIA_LOAD_PATH
- PLOTS_DEFAULT_BACKEND

REPL

```
varinfo()                # inspect vars in namespace
```

Scripts

```
include(myfile.jl)       # run script
#!/usr/bin/julia         # shell she-bang
```

2 Basic Syntax

```
x = 0                    # simple assignment
global x += 1            # variable scoping
```

Conditionals

```
if 1==1                  # if w/o paren
    ...                  # logic
end                      # condit'ls all "end"
if <cnd>; <stmt>; end    # single line
z = <cnd> ? <exp1> : <exp2>    # ternary
for <cnd>; <stmt>*; end  # for syntax
for i=1:5,j=1:10         # multi-condition; ...
```

Functions

Functions do not exist as methods on "receiver" objects, but rather pass all objects / values as parameters. Polymorphism is accomplished at run-time through "multiple dispatch", as determined by passed parameter types. Inquire into polymorphic overrides with `methods(myFn)` call.

```
function f(x,y)
   println("Sum: $x, $y")  # side-effects
   if(x+y>100) return 5    # explicit return
   x + y                   # implicit return
end
f(x,y) = x + y             # ≈ to above
g = f;                     # functions as variables
(x) -> x+a                 # anonymous function
f(x,dim=2)                 # named arguments
f(x,c...)                  # spread operator
∑(x,y) = x + y             # unicode in fn names
```

By convention, exclamation indicates call by reference on 1st argument.

```
f!(rcvr,<arg>*);           # receiver free to mutate
```

Operators

```
+(1,2,3)                   # operators are fns
>>  <<                     # logical shifts
>>>  <<<                   # arithm. shift
⊻                          # \xor | \veebar
==                         # comparison
===                        # object equality
```

Higher Order Functions

```
filter(z -> z>3, x)        # filter
map(z -> z^2, [1,2,3])     # map
broadcast(myFn,myArray)    # ≈ map
f = x -> x^2               # anonymous fn
myFn.(myArray)             # sugar for broadcast
reduce(+, xs, 5)           # 5 + ∑_i xs_i
```

Comments

```
\# comment text            # single line
\#= outer                  # multi-line
    \#= inner ...          # nestable
=\# outer =\#              # end of comments
```

Exceptions

```
try ...; catch e; end; finally ...;
```

Macros

```
@time mean(y)              # time mean fn
@code_llvm f(1)            # look at asmb
@show cdf(Normal(0,1),.5)  # evaluate
@which copy([1,2,3])       # inspect mult dispatch
@benchmark fn()            # basic code profile
@profile fn()              # n simulation runs
@assert                    # assert
@debug                     # debug
```

3 Collections

Arrays

∃ *numerous ways to create:*

```
A = [1 2 3 ...]            # concrete
A = [1:1000]               # comprehension sugar
A = [1 2; 3 4]             # matrix-style
A = j:k:n                  # lazy init, step k
[f(i) for i in 1:10]       # comprehension
vcat(x,y,z)                # = [x;y;z]
hcat(x,y,z)                # = [x y a]
```

Incremental creation using:

push! pop! append! prepend!
deleteat! pushfirst! fill! insert!

Array assignment is only a reference. Use `copy()` or `deepcopy()` to make a 1-layer, or full copy, respectively.

```
B = A                      # reference-only
B = copy(A)                # shallow copy
```

Homogeneous arrays (same type) are fast; heterogeneous arrays (::Any) are possible. Can also create a Union of enumerated types:

```
B = Union{Int64,Float64}[..]
```

Array inquiry functions:

sort isempty findall
eltype collect size

Operate over arrays with:

sum product any all
minimum maximum findmin findmax
first last count getindex
filter map reduce mapreduce

Transform arrays with:

splice! reverse! sort! zip

Strings

- `s1*s2` # concatenate s1, s2
- `s^n` # repeat s n times
- `$(var)` # interpolation

Helper operations:

join replace [l|r]pad [l|r]strip
search rsearch in
index rindex beginswith endswith
isalnum isalpha isascii isblank
isdigit isgraph islower isprint
isspace ispunct isupper isxdigit

Dictionaries

Dictionaries are mutable and type-unstable if different types are injected. Basic syntax:

```
myDict = Dict(             # declare ...
    'a'=> 1                # assign ...
    'b'=> 2 )
[d(i)=value for (i,value)] # comprehension
```

Helper methods:

get getkey values keys
collect haskey in
length delete! pop! merge

Tuples

```
a = (1,2,3)                # creating
a = tuple(1,2,3)           # ditto
a = ntuple(n,f)            # function gen
a, b = (1,2)               # destructuring
tup = (a=1,b=2)            # "named" tuples
tup[1]; tup[2]             # indexing
first(tup); last(tup)      # ibid
tup.a                      # dot-notation (≡ above)
```

Tuples are immutable. ∃ the following helpers:

values keys pairs collect

Sets

```
s = Set([1,2,3,...])       # creation
i = IntSet([1,2,3,...])    # sorted ints
intersect(s1,s2)           # s1 ∧ s2
union(s1,s2)               # s1 ∨ s2
setdiff(s1,s2)             # s1 ¬ s2
symdiff(s1,s2)             # s1 ⊕ s2
issubset(s1,s2)            # s1 ⊂ s2
```

Additional helper methods:

add! complement!

4 Types

Missing

- missing::Missing
- nothing::Nothing
- NaN::Float64
- Inf

Numeric Types

- Int64 42
- Float64 ..... 0.2, 1e10, 4.
- Char ..... 'a','b'
- Bool ..... true, false
- Complex ..... 5-2im, complex(5,2)

Basic math functions:

abs cmp round divrem
real imag

Structs

```
mutable struct MyS{T<:Number}
    property1              # untyped
    property2::String      # concrete type
    property3::T           # type constraint
end                        #
mutable struct MyMutS ...  # mutable
s = MyS("a","b",5)         # initialization
a = s.property3            # referencing
```

To define abstract types:

```
abstract type MyGenType end
abstract type MyConcType <:MyGenType end
```

Regexs

```
rm = match(r"regex",s,i)   # execute
    rm.match               # substring matched
    rm.captures            # tuple of matches
    rm.offsets             # vector of matches
```

Conversion

```
parse(Float64, "3.14")     # float from string
float64("3.14")            # ditto
string(3.14)               # inverse
int8("123")                # int from string
hex(x); oct(x); dec(x)     # various casts
```

Type System & Generics

```
q::Number                  # type annotation
f{T<:Number}(x::T,y::T)    # parametric types
f{A<:B}(x::A)              # subtype constraints
```

Type System Helper Methods

- subtypes(type)
- supertype(type)
- fieldnames(type)
- isa(field,type)
- typeof(obj)
- isequal(x,y)

5 File I/O

open() a file, returning a handler (passing a "modality" ∈ read, write, append); then close.

```
h = open("f.txt","r")      # create handle
cont = read(f,String)      # read
close(h)                   # close
```

It is, however, idiomatic to use the "do" construct:

```
open("f.txt","r") do h     # create
    cont = read(h,String)  # read
```

```
end                          # implied close
```

Reading line by line:

```
open(...) do h               #
   for ln in eachline(h)     #
      println(ln)            #
   end                       #
end                          #
```

Writing:

```
open("f.txt","w") do h       #
    write(h,"text\n")        #
end                          #
```

Helper methods:

position	seek	seekstart	seekend
skip	isopen	oef	isreadonly
ltoh	ltol	[de]serialize	download
readbytes	readcsv	readall	readlines

6 Linear Algebra

Building Matrices

```
[1. 2. 3.; 4. 5. 6.]         # 2x3 matrix ∈ ℚ
A = [1 2 3]'                 # transpose
transpose(A)                 # ibid
reshape(A,dims)              # transform
A = ones(2, 2)               # 2x2 matrix
A = zeros(2,2)               # 2x2 0 – matrix
A = Diagonal(A)              # diag. of A
A = I                  # Identity matrix
A = Matrix{Int}(I, 3, 3)     # ibid, 3x3 of ints
A = reshape(1:10,5,2)        # shape from linear
C = similar(A)               # same dims
```

Indexing into Matrices

```
A[2,2]                       # access element
A[1:4,:]                     # access rows
A[:,1:4]                     # access cols

A[[1,2,4],:]                 # deselect row
diag(A)                      # retrieve diagonal
size(A)                      # get dimensions
```

Matrix Math

```
eigvals(A)                   # eigenvalues
eigvect(A)                   # eigenvectors
inv(A)                       # inverse
det(A)                       # determinant
A .* B                       # element mult
A * B                        # matrix mult
dot(v1,v2)                   # vector dot prod
A\b                          # solve Ax = b
rref(A)                      # red. row-echelon
nullspace(A)                 # nullspace
```

7 Statistics & Probability

Import "Statistics", "BaseStats", "Distributions".

Random Numbers

```
A = rand(2)                  # 2 rand floats
A = rand(2,2)                # 2x2 matrix ∈ ℚ
rand(Uniform(a,b),2,3)       # from distribution
```

Array / Matrix Reducers

```
sum(A,dims=2)          # sum rows (dim=2)
max(A,dims=1)          # max by cols (dim=1)
min(A,dims=2)          # min by rows
cumsum(A,dims=2)           # cumulative ∑
accumulate(max,A,dims=1)  # apply max fn
```

Probability Distributions

```
D = Normal(0,1)              # a std normal
quantile(D,[.9,.95])         # α = [.1,.05] quantiles
cdf(D,x)                     # F_{μ,σ²}(x) (here Φ(x))
Binomial(.5)                 # other distrib'ns
fit(D,x)                     # generic fit
fit_mle(D, x)                # θ̂^MLE estimation
```

∃ many dists, eg: uni- & multivariate, & matrix.

Statistics

```
var(Normal())                # var(𝒩(0,1))
var([1,2,3])                 # S̃(x⃗) (here = 1)
mean([1,2,3])                # mean, here = 2
mod = @formula(y ~x)# modelling (see here)
lm(mod,data)                 # regression
```

max	mean	skewn'ss	cov
min	var	kurtosis	invcov
extreme	std	mgf	location
cor	mode	pdfsq.n'rm	scale
	cf		

8 Differential Equations

Import and use "DifferentialEquations". Then:

- Define the problem (system, tspan, ICs)
- Solve the problem (parameterize solver)
- Analyze the solution (plot or inspect data)

```
f(t,u) = 1.01*u              # the system
u0=1/2                       # initial condition
tspan = (0.0,1.0)            # timespan
prob = ODEProblem(f,u0,tspan) # wrap
sol = solve(prob)            # solve
sol = solve(prob,reltol=1e-6)# parameterize
sol = solve(prob,Tsit5())    # different solver
sol[5]                       # 5th step value
sol.t[3]                     # 3rd timestep val
sol(.45)                     # interpolated val
plot(sol)                    # plot
```

Systems of Equations

```
function lorenz(t,u,du)      # lorentz eqn
 du[1] = 10.0(u[2]-u[1])
 du[2] = u[1]*(28.0-u[3]) - u[2]
 du[3] = u[1]*u[2] - (8/3)*u[3]
end                  # now pass to ODEProblem()
```

9 Computer Algebra

```
ENV["PYTHON"]="" # use private Python
Pkg.add("PyCall"); Pkg.build("PyCall"); using PyCall
```

then reload Julia. See here for a notebook, here for a julia tutorial, or here for Python.

```
Pkg.add("SymPy");    using SymPy
```

```
x,y,z = symbols("x y z")     # define symbols
 vars x,y,z                  # equiv to above
expr = x+2*y                 # an expression
expand(x*(x+2*y))            # expanding
factor(x^2+2*x*y)            # factoring
simplify(f-g)                # reduce
cancel((x^2 + 2*x + 1)/(x^2 + x)) # std form
apart(expr)              # partial frac. decomp.
f=(x+1)^2;g = x^2-2x+1       # functions
f-g                          # higher order
expr(x=>1,y=>2)              # parameterize
expr.subs([(x,2), (y,2)])    # ibid
```

Solving

```
solve(expr,y)                # solve: expr=0
eqn = Eq(expr2,2)            # equation object
solve(eqn)                   # solve arb. eqn
```

Calculus

```
diff(cos(x), x)              # differentiate wrt x
diff(x^4, x, x, x)           # d³/dx³(x⁴)
integrate(cos(x), x)         # indefinite
integrate(exp(-x), (x, 0, oo))# definite: ∫₀^∞ eˣdx
sympy.Integral(cos(x)^2, (x, 0, PI))  # ¬ eval'd
limit(sin(x)/x, x, 0)        # limit
exp(sin(x)).series(x, 0, 4) # series expand
```

Linear Algebra

```
M = Sym[1 2 3; 3 2 1]        # matrix
N = Sym[0, 1, 1]             # vector
M*N                          # symbolic eval
det([1 x; y 4])              # LA lib. works
```

Miscellaneous

```
Or(x,y)                      # logic
sympy.expand_trig(sin(2x)+cos(2x))# trig
trigsimp(sin(x)^2 + cos(x)^2)    # simplify
powsimp(x^a*x^b)             # power simplify
```

10 Packages & Modules

Installing, Managing

Can alternatively use "package" mode, hitting "]" at REPL, then "?" for commands, which include: add, update, status, rm, etc.

```
Using Pkg                    #
Pkg.add("name")              #
Pkg.clone("https://github...")#
Pkg.checkout(url)            #
```

After adding a package, you must explicitly include it with either `import` *or* `using`:

```
import Plots                 # include (clean NS)
using Plots                  # ibid (cluttered NS)
require(file)                # load once
reload(file)                 # reload
include(file)                # set dir, load
```

Common Packages

- GLM
- DataFrames — Tabular data manipulation
- NullableArrays
- MixedModels
- Optim
- Distributions
- JuMP — Machine Learning
- SymPy — Symbolic computation
- Convex
- Losses
- Transformations
- jplyr
- Query
- SciKitLearn
- PyCall
- RCall
- Roots — [Transcendental] Eqn Solver
- CSV
- ODBC — Database connectivity
- Mamba
- HTTP
- Weave — Document generation
- OdsIO — Open Documents
- Images

11 Pluto

Basic Use

Keyboard Shortcuts

Try customizing with this.

- Ctl + Alt + b ?
- Ctl + Alt + l
- Ctl + Alt + v toggle cell visibility
- Ctl + Alt + s split cell
- Ctl + Alt + enter new cell above

UI and HTML