Julia Reference Card v0.9

1 Variable Types

1.0 On All Objects or Collections (c)

`is(a, b) or ===`	object identity
`isequal(x,y) or ==`	value identity
`isa(x, type)`	test if x is a type
`isless(x,y)`	consistent x<y test
`typeof(x)`	get x's concrete type
`tuple(x's)  tuple([]...)`	create tuple
`ntuple(n, f::Function)`	tuple f(i) for i:n
`object_id(x) ; hash(x)`	id equiv to === & ==
`copy(x) ; deepcopy(x)`	shallow or recursive copies
`eltype(d)`	type of elements
`eval ; evalfile`	evaluate expression
`collect(c) ; (c...)`	array of all items with k,v tuples for dicts
`s=[c,x] ; s=vcat(c,x)`	Add to s vertically
`s=[c x] ; s=hcat(c,x)`	Add horizontally
`hvcat(a,r,c) ; [a b;c d;..]`	Concatenate r+c
`empty!(c) ; isempty(c)`	empty or test c
`x in s; in(s,x);  !in(s,x)`	is x a member of s
`length(c)  ; endof(c)`	length and last index of c
`size(c[,d])`	size of c in dimension d
`sum(c[,d]) ; prod(c[,d])`	fn over dimensions
`fill!(c,x)`	fill A with value x
`minimum(c[,d]); maximum`	of seq or array (with dims)
`findmin(c) ; findmax`	returns max and index
`any(c[,dims]) ; all`	boolean tests
`count(f(x),x)`	num where f(x) is true
`first(c) ; last(c)`	O(1) first or last element
`getindex(c,i)`	get value at index i
`unique(c)`	ordered array of uniques
`filter(f(x),c) or f(k,v)`	return items where true
`filter!(f(x),c)`	or update collection
`map(f,c) map!(f,c)`	transformations
`reduce(op,v0,c) ;`	reduce with operator
`mapreduce(f,op,c)`	from init value v0

1.1 Numbers

`42 0x2A 0o52 0b101010`	42 (dec,hex,oct,bin,)
`0.2 .8 4. 1e10 1e-7 3.2f0`	floating point value
`Inf NaN`	
`z = 5 − 2im`	complex number
`z = complex(real, imag)`	complex number
`real(z); imag(z)`	real and imag part of z
`2//3 - 1//2`	rational numbers (gcd)
`true; false`	boolean constants
`abs(n)`	absolute value of n
`divrem(x, y)`	(x/y, x%y)
`cmp(x,y)`	x<y: -1, x==y: 0, x>y: 1
`42 0x2A 0o52 0b101010`	42 (dec,hex,oct,bin,)
`round(x,n)`	round x to n dec places
`int() int8() to int128()`	int from string or num
`float("3.14")  float16()`	float from string or num
`float32() float64()`	
`string(3.14)`	conversion
`hex(n)  dec(n)  oct(n)`	create hex, dec, oct,
`base(b,n)`	base b string
`int('x')`	code point of char

1.2 Sequences (arrays are mutable, tuples and strings are immutable). 1 dimensional arrays (column) replace vectors and arrays are indexed from 1 to end. Arrays use [] but heterogeneous arrays, cells, use {} and can replace lists.

`a=l=[1, 2, 3, 4] or []`	create 1 dim Array
`s=l={1,"ba",{1+2im,1.4}, 4}`	list or 1d cell creation
`s=t=(1,"ba",[1+2J,1.4], 4)`	tuple creation
`s=l=linspace(start, stop, n)`	n items between start & ...
`l=[t...] ; t=tuple(l...)`	list / tuple conversion
`s=1:1000`	range of integers
`a=[1:1000]`	1d array of ints
`s[3][1]`	get element (1+2im)
`l[end-1][end]`	get element (1.4)
`s[i:j]; s[i:]; s[:j]`	slicing (i & j inclusive)
`s[i:k:j]`	slice with stride k
`s[2:2:]; s[end:-1:1]`	every 2nd ; reverses
`l[i:j]=['a','b','c','d']`	replace equal slice
`push!(a,x)  x = pop!(a)`	Add/remove end of a
`unshift!(a,x)  x = shift!(a)`	Add/remove start of a
`append!(l,l2) ; prepend`	append l2 at end of l
`insert!(l,i,x)`	insert x at pos. i
`splice(a,i:j[,newarray])`	remove i to j
`reverse!(1,i,j)`	reverse l from i to j
`sort!(l)`	sort (many options)
`zip(s,t,...)`	[(s[0],t[0],...),..]

1.3 Dictionaries

`d={'x'=>42,'y'=>3.14,'z'=>7}`	dict creation
`d={i => f(i) for i=1:n}`	using comprehension
`[] for inferred types`	{} for any type
`d['x']`	get entry for key 'x'
`length(d)`	number of keys
`delete!(d,'x')`	delete entry from dict
`has_key(d, k)`	does key exist?
`keys(d)`	iter of all keys
`values(d)`	iter of all values
`collect(d)`	array of keys / values
`get(d,k,x)`	get value, default x
`getkey(d,k,x)`	get key, default to x
`merge(dict, ...)`	merge dicts
`pop!(d,k,x)`	return & delete item

1.4 Sets

`s=Set(s...)`	create set
`s=IntSet(i...)`	create sorted int set
`add!(s,key)`	add an element
`issubset(s,t); s<=t`	all s in t?
`union!(s,t)`	array if t is array
`intersect(s,t)`	elements in s and t
`setdiff!(s,t) | (s,c)`	all s not in t
`symdiff!(s,t) | (s,n) (s,c)`	all either s excl or t
`complement!(s)`	set-complement intset

1.5 Strings and Regular Expressions

`"bla"; 'hello "world"'`	string (of bytes)
`\\`	backslash
`\N{id} \uhhhh \Uhhhhhhhh`	unicode char
`\xhh`	hex
`'\u78' '\u2200'  '\U10ffff'`	unicode string
`@sprintf("%Fmt", args...)`	string formatting
`%s  %03d  %.2f  %+.0e  %E`	string, int 3char + lead zero, float 2 precision
`t="eat" ; "$t here"`	var interpolation
`s*s ; *(s,s1,s2)`	concatenate strings
`s^n ; ^(s,n)`	repeat s n times
`join((s,s,s),sep)`	join string with separator
`collect(s)`	return an array from
`utf8(s)`	s to utf-8 string
`char(i)`	char from code point

Other String Methods:

search & replace: `search(s,pat,i), rsearch(s,pat,i), in(pat,s) index(s,pat,i), rindex(s,pat,i), beginswith(s,pat), endswith(s,pat), replace(string, pat, r[, n])`

formatting: `lowercase, uppercase, ucfirst, lcfirst`

splitting: `split(s,m), rsplit(s,m), chop, chomp`

padding: `lpad(s,n,p), rpad(s,n,p), lstrip(s,c), rstrip(s,c), strip(s,c)`

checking: `isalnum, isalpha, isascii, isblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit`

Regexes:

`rm=match(r"regex",s,i)`	1st. nothing if no match
`rm.match`	substring matched
`rm.captures`	tuple of matches
`rm.offset`	offset to match
`rm.offsets`	vector of offsets
`matchall(r"",s) -> [s s ...]`	vector of matches
`eachmatch(r"",s[,o]) -> iter`	iterator over matches
`flags after the double quote`	
`i`	case insensitive
`m`	multiline string
`s`	single line string
`x`	ignore whitespace

1.6 Arrays (homogeneous & type may be specified)

```
Array(T, dims)                  Uninitalized dense dim
                                array of Type T
```

Initialize different arrays (sometimes with T else just dims):

```
  zeros ones  trues  falses  rand randdf  randn
  eye   eye(n)  linspace(start, stop, n)
```

Vector = 1 dim column array or cell is like a list in Python
Functions on arrays:

```
  nnz (num non zero values) stride(A,n)  strides(A)
  ndims, transpose & ctranspose .' & '
cell(dims ...)                  uninitialized
                                heterogeneous array
Array(Int32,0)  []  {}          Empty array or cell
reshape(A, dims)                new shape, same data
similar   reinterpret
[a,x] [a x]  {c,x} {c x}        add element
a = [f(x,y,...) for x=rx,       array comprehensions
y=ry, ...]
```

1.7 DataFrames (using DataFrames)

```
DataArray  NA  NAtype           Array with missing values
DataFrame(A=1:4,B=[...])        Tabular hetero dataset
removeNA  replaceNA(dv,val)     remove or replace NAs
  failNA
df[2,"A"]  df[[rows,]1:2]       get & slice
df[1:2,["A","B"]]
df[df["A"] % 2 .== 0, :]
colnames!(df[,newnames])        read or insert col names
head, tail, describe
join(df..., jointype)           join two dataframes
groupby(df,catvar)              split df by categorical var
by(df,catvar, df->f(df[]))      split and apply fn or
by(df,catvar,:(n=...; m=...))   expression to subset
stack(df,categorical var)       reshape data
readtable(fname,header=false,   defaults are true, comma
  separator='\t')               also, writetable(f, fname)
```

2 Basic Syntax

```
if expr statements              Conditional
elseif expr statements          ; if on same line
else statements end             terminate with end
z = cond ? x : y                ternary version
z = ifelse(cond, x, y)          as ? but all args evaluated
z = cond && x                   short circuit

while expr statements end       while loop
while true .. if cond break     do .. while equivalent
for target in iter              for loop
  statements ; end
for i=itr, j=itr ... end        over multiple variables
for key in keys(d)...           over dictionary
break, continue                 end loop / jump to next
s=start(I);while !done(I,s)     iterator from sequence
  (i,s) = next(I,s)
print("hello world")            print or println (new line)
[expr for x in seq lc ]         list comprehension
nothing                         empty statement
```

```
function f(params) ... end      function definition
f(x, y=0) = return x+y          optional parameter
f(a,b,c...) =                   varargs c = [] or ()
f(a,b; dir="nth") =             named args
f(a,b; d=5,e...) =              varargs as of k,v tuples
... |> (x,f)                    apply fn to preceding args
f(1,1), f(2), f(y=3, x=4)       function calls
(x) -> x+a                      anonymous function

function make_adder_2(a)        closure, alternatively,
  add(b) = return a+b             function mkadr(a)
  return add                        b -> a+b
end                               end
let v=1,w=" "... ; end          scope block with vars
@time()       gc_disable()      report time elapsed
@profile     Profile.print      profile, print & clear

global v                        bind to global variable
: ... ; quote ... end           create an expression
eval(expr)                      evaluate expression

using name    fn()              load module namespace
import name   name.fn()         import gives named access
require(filepath)               Load file once
reload(filepath)                 and reload it
include(filepath)               set dir & load source
evalfile(file)                  execute file
cd("data") do                   safely write file in a
  open("outfile", "w") do f     directory and close after.
    write(f,data)
  end
end
```

4 Exception Handling & Debugging

```
try ...                         Try-block
catch [y]                       catch exception as var
    print data
    error("...")                exception handling
end
finally ...                     in any case
@assert expression              debug assertion
throw(e)                        explicit expection
using Debug                     loads the debugger
@debug  @bp                     before module, set breakpt
  l   p var1, ...               list lines, print vars
  s   c                         step into, continue to @bp
```

5 System Interaction

```
run(`cmd`) or ;cmd              system call
spawn(`cmd`)                    run asynchronously
success(`cmd`)                  bool for exit condition
process_running(process)        determine if running
process_exited(process)         determine if has exited
kill(process, signum)
readsfrom(command)              (its stdout, process)
writesto(command)               runs asynch & returns
                                 (its stdin, process)
readsandwrite(command)          (its in, its out, process)
detach(command)                 run & outlive Julia process
setenv(command,env)             set vars for running
ENV  EnvHash->EnvHash           Sys environment vars
getpid()                        get Julias pid
clipboard(x)                    print x to clipboard
s = clipboard()                 or s from clipboard
@time()  @elapsed()->secs       time and expression
strftime([f,]time())            time as string
cd(f[,dir])                     run f in temporary dir
```

Filesystem Operations

```
gethostname(), getipaddr(), pwd(), cd('dir'),
mkdir(p,mode), mkpath(p,mode), rmdir(p),
ignorestatus(cmd),
```

redirect in run commands: `|> std output, |>> append stdout, |.> stderr  to process, file or DevNull`

6 Input/Output

```
open(filename, mode)            open file (a & w create,
mode = r r+ w w+ a a+            + is both r&w, w truncates)
open(f(),args)                  f(result of open args)
close(stream)                   flush and close
write(stream, x)                write binary x to stream
writedlm writecsv               array, dlm with csv delimeter
read(stream, type[,dims])       read value from stream
readbytes readdlm readcsv       nb bytes, array, csv
readall readline[s]             all as string, line or lines
position(s)                     get position of a stream
seek(s, pos)                    seek stream to position
seekstart(s),  seekend(s)        to start  to end
skip(s, offset)                 seek relative
isopen  eof  isreadonly         open? end of file?  read only?
ltoh(x)  htol(x)                little endian conversions
[de]serialize(stream,val)
download(url[,localfile])       unix download
 + others
```

7 Areas Not Covered

Julia has a dynamic type system but with a rich language of types including parametric. Type declaration is optional so the casual user can usually ignore it.
Multiple dispatch permits methods to be called based on the types of all unnamed arguments.
Object orientated design can be achieved by combining type definition and multiple dispatch to associate methods with new classes of objects
Tasks or Coroutines permit computations to be flexibly suspended and resumed, effectively enhanced generators.
Parallel and distributed computing and metaprogramming are supported.

```
help(name)                      get help on object
apropos("search string")        search docs for string
```