Julia Cheat Sheet

Mo D Jabeen

February 14, 2023

Contents

1 General

Listing 1: Function definition.

```
function name()
code
end

function name()
r=3

r,r+2 #Omit the return keyword for tuple return
end
```

- printf for formatted prints uses the module Printf and is macro with synatax @printf
- %3f : used to show 3 sig fig
- ë : scientific notation

- index starts at 1 :O
- Strings can be indexed like arrays
- Combine strings using *
- try, catch : for error handling

Listing 2: Dict definition.

```
d = Dict(1=>"one", 2 => "two")
d[3] = "three" # Add to the dict

#Loops and funcs can also be placed in dicts
```

Listing 3: Loop/arrays definition.

```
for i in 1:5 # This calls the iterate func
println(i)
end

a = collect(1:20) # convert into an array

a = map((x) -> x^2, [1,3,5,3]) # map performs func on each array element

foreach(func, collection) #operate func on each val of collection
```

Listing 4: Struct definition.

```

mutable struct name
        string::AbstractString
        boolean::Bool
        age::Int
        a::Array{Int,5}
end

```

```
9         newstruct = name(...)
10
11        # Internal constructors are used to place constraints on the code
12
13        mutable struct name
14                meh::AbstractString
15                numb::Int
16
17                name(blah::AbstractString)= new(meh,4)
18                # this enforces if a struct without
19                #a number is given 4 is placed
20        end
```

Listing 5: Tenancy operations

```
1
2         x > 0 ? 1 : -1
3         # If the condition is true 1 is returned else -1 is
```

- Avoid globals
- Locals scope is defined by code blocks ie func, loop not if
- Built in funcs such as iterate can be extended via multi-dispatch
- Use the Profiling package for measuring performance.

2 Objects/Methods

Structs mainly used to create new data type objects.
Inner and outer constructor methods for structs define how a new object is created based on data input.
Inner constructors enforce the same checks for multiple data types.

Listing 6: Constructors

```
1
2       struct name{T<:Integer} <: Real
3       # <: shows all values are included in that set
4       # {for arg} outer for object
5
6               num::T
7               den::T #ensures both are of type T
8
9               #Function checks if the input numbers are empty for every object
10              function name{T}(num::T, den::T) where T <: Integer
11
12                      if num == 0 && den == 0
13                              error("invalid")
14                      end
15                      new(num,den)
16              end
17      end
18
19
20      name(n::Int, d::Float) = name(promote(n,d)...)
21      #Outer constructor
22      #Promote converts values of a single type to the same type
23      #choosing the type to work with both
24
25
26      # MULTI-DISPATCH FUNCTION
27
28      function blah(n::Int, d::Int) = println('meh')
29
30      function blah(x::Int, y::name) = println(x*y.num)
31      #This func now has two methods (multi dispatch)
32
33      # The T is the type inside the vector which refed in the func
34      function myfunc(x::Vector{T}) where T
35              b::Vector{T} = x.*3
36      end
```

3 Modules

Modules allow for better namespace control and cleaner structure.

They are not attached to a file, can have multiple modules in a file and multi files for the same module.

using modulename: Includes all code and exported variables.
import modulename: Includes only the code.
Can use submodules which are accessed via . operator.

4 Differences from Python

- Use immutable Vector (same data type) instead of arrays (python would use list)
- Indents start with 1
- Include end when slicing ie [1:end] not [1:]
- Use [start;stop;step] format
- Matrix indexing creates submatrix not tuple ie X[[1:2][2:3]]
- To create a tuple from a matrix use (like python) X[CartesianIndex(1,1), CartesianIndex(2,3)]
- Variable assignment is not pointer assignment ie a= b creates new variable so they remain separate.
- push! is the same as append
- % is remainder not modulus
- Int is not an unknown size its int32
- nothing instead of null

5 Metaprogramming

Julia code is represented after compiling as a data struct of type Expr.

$: Used as interpolation for literal expression in a macro.
eval: Executes the code from Expr data type.
: Turns code into an expression (can also used quote for blocks)
Can use Expr data types as inputs to functions.

5.1 Macros

Compiled code as an expression not executed on runtime but during parsing.

Listing 7: Macro definition

```
1
2       macro name()
3
4       end
5
6       @name() # Run using the @ operator.
```

Macros are used in code when an expression is required in multiple places before it is evaluated.

Listing 8: Create code

```
1
2       struct MyNumber
3       x::Float64
4       end
5       # output
6
7       for op = (:sin, :cos, :tan, :log, :exp)
8       @eval Base.$op(a::MyNumber) = MyNumber($op(a.x))
9       end
```

6 Concurrency

Julia combines multi threads and cores using the same memory space as threading. CPUs using separate memory spaces are defined as multi-processor or distributed computing.

mutex: Single lock mechanism for controlling accessing to data.
semaphore: Value signifying what are the resources being used on, for process synchronization.
Julia code tends to be purely functional and avoids mutation, generally opting for only local mutation.
If there is a shared states locks should be used or a local state (an object shared by all threads.)
A shared local state gives higher performance.

6.1 Asynchronous

6.1.1 What are Tasks ?

Tasks are used for asynchronous calls, ie waiting for external signals. Tasks allow switching at any point in the execution between them and don't use extra memory space (call stack).
wait(t) - waits for the tak

Listing 9: Async functions

```
1
2         t = @task func()
3         OR
4         t= @task begin ... end
5         OR
6         t = Task(func)
7
8         schedule(t,[val],error) # Allocate task to scheduler, pass val
9
10        # if error true, val passed as an exception
11
12        @async func() same as schedule(@task func())
13
14        asyncmap(func,collection,ntask,batch)
15
16        # Return collection with the func executed on by ntasks.
17        # Batch executes on collection in groups set by number of batch.
18
19        yield() #Switch to scheduler to allow another task to run
20        yield(t) #Switch to task t.
21
22        Condition() #Edge triggered event source
23        thread.condition – thread safe version
24
25        Event() #Level triggered event source
26
27        notify(condition,val,all,error) #Wake up tasks waiting for condition
28
29        semaphore(sem_size) #counting with max at semsize
30
```

```
31          acquire(s) #get a semaphore, blocks if none available
32          release(s)
33
34          #Use below format for locking
35
36          lock()
37          try
38          ...
39          finally
40          unlock()
41          end
42
43          bind(channel, task)
```

Listing 10: Async wait functions

```
1
2           wait([x])
3
4           x {
5                   Channel: Wait for val
6                   Condition: Wait for notify
7                   Process: wait for process to exit
8                   Task: wait for task to finish
9                   RawFD: change the file descriptor
10          }
11
12          #if there is no x will wait for schedule to be called
```

6.1.2 What are Channels?

Channels are a first in first out queue, used to connect tasks in a memory/race safe way. They can be bounded to a task, by being placed as a parameter and therefore do not need closing.

Listing 11: Channel Functions

```
1
2           c = Channel{Type}(limit) #limit is max number of objects in queue
3
4           put!(channel,data) # Place data into channel
5           take!(channel) #Read data from channel
6
7           Channel(func()) – Bind a channel with a task
```

- Readers will block on a take if the channel is empty
- Writers will block on a put if the channel is full
- Wait will wait until the channel has data
- isready test if the channel has data

6.2 Multi threads

Use atomic vars to ensure expected correct operation when using threads (ie for arithmetics). Careful of finalization (tasks to clean up before garbage collection.)

Listing 12: threading Functions

```
1
2         Threads.@threads [schedule] for ... end
3
4         [schedule] {
5                 default: :dyanmic assumes equal load per thread, cant
6                 guarantee thread id on an iteration
7                 :static one task for thread, can guarantee same id for an
8                 iteration
9         }
10
11        threads.foreach(f,c,ntasks) #operate function on channel with
12        n threads
13
14        Threads.@spawn func() #Create task and schedule to run on any
15        available thread
```

7 Logging

Inserting a logging statement creates an event, logging is allows for better control and visibility than print statements.

Listing 13: Logging Functions

```
1
2       @debug #Auto set to not output to stderr
3       @info # mid level
4       @warn #Higher level
5       @error #highest level, generally not needed
6       # (use exceptions instead)
7
8       @__ msg var x=var y=func() #msg in markdown
9
10      @__ "blah $var "
```

7.1 How do you process a log ?

Log creation and processing are separate to allow for module level and app level work.

Listing 14: Logger

```
1
2       ConsoleLogger([steam,] min_level=Info) #stream can be a file io
3       SimpleLogger([stream,] min_level=Info)
4
5       global_logger(logger)
6       with_logger(logger) do ... end #Local logger
```

Listing 15: Log filter

```
1
2       disable_logging(level) #disable below this level
3       should_log(logger,level) #return true if accepts this level
4
5       handle_message(logger,level,message,val ...)
```

Listing 16: Example log

```
1
2       # Open a textfile for writing
3       io = open("log.txt", "w+")
4       IOStream(<file log.txt>)
5
6       # Create a simple logger
7       logger = SimpleLogger(io)
8       SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())
9
10      # Log a task-specific message
11      with_logger(logger) do
12                      @info("a context specific log message")
```

```
13                  end
14
15          # Write all buffered messages to the file
16          flush(io)
17
18          # Set the global logger to logger
19          global_logger(logger)
20          SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())
21
22          # This message will now also be written to the file
23          @info("a global log message")
24
25          # Close the file
26          close(io)
27
28          # Create a ConsoleLogger that prints any log
29          #messages with level >= Debug to stderr
30          debuglogger = ConsoleLogger(stderr, Logging.Debug)
31
32          # Enable debuglogger for a task
33          with_logger(debuglogger) do
34                      @debug "a context specific log message"
35              end
36
37          # Set the global logger
38          global_logger(debuglogger)
```

8 Packages

Designed around environments which can be local to inviduals or projects. Allows exact set of packages and their version in an environment to be controlled and repeated. All tracked in the manifest file.
A better version of python virtualenv.

Listing 17: Pkg commands

```
1
2         activate [project name]
3
4         develop --local [package name] #Allows you to use a local version of a
5         # package to develop
6
7         free [package name] #Go back to the registered version
8
9         get "url" or "local path"
10
11        pin [package name] #Never update
12
13        # Activate another persons project
14        activate .
15        instantiate
```

9 Plotting

The Julia plot pkg used is **Gadfly**, git: https://github.com/GiovineItalia/Gadfly.jl.

Used for plotting and visualization, using the browser to supply interactive plots, also supports svg,png,postscript and pdf (done via draw command).

9.1 General

Listing 18: Plot commands

```
1
2         plot(data, elements ....; mapping)
3         #data is a dataframe
4         #elements are the plot ie x= :col, color = blue etc
5         #mapping is the type of graph ie Geom.point, Geom.line
6
7         map(display, plot)
8         #Or can use REPL to output to default media viewer (browser)
9
10        plot(func(), lower, upper, elements; mapping)
11        #lower upper are x bounds
12
13        plot(Vector{funcs()}, lower, upper, elements; mapping)
14        plot(func(), xmin, xmax, ymin, ymax, elements; mapping)
15
16        # Can also add elements to plot via push!
```

To output the plot as a html file, run code in REPL using include("filename.jl").

Widerform is when cols use the sames measurment type but are grouped in cols.

Elements have a Scale (continous or discrete) and Guide class:

9.2 Aesthetics

Data is binded to aesthetics, defined by a number of elements: Scales, Coordinates, Guides and Geometrics.

9.2.1 Themes

Allow easy setting of a bunch of parameters, the theme overrides the default values.

Style will override the current theme.

9.2.2 Geometries

Are the types of graph: Point, line, bar, candle etc.

9.2.3 Scale

Scale allows a characteristic to be used to show the different grouping of values ie alpha(transparency),color, shape, linetype.

Listing 19: Scale examples for grouping

```
1
2         plot(x=xdata, y=ydata, color=cdata,
3         Scale.color_continuous(minvalue=-1, maxvalue=1))
4
5         plot(x=x, y=y, color=x+y, Geom.rectbin,
6     Scale.color_continuous(colormap=p->RGB(0,p,0)))
7
8         plot(x=xdata, y=ydata, color=repeat([1,2,3],
9         outer=[4]), Scale.color_discrete)
10
11        plot(x=rand(12), y=rand(12), color=repeat(["a","b","c"],
12         outer=[4]),
13     Scale.color_discrete_manual("red","purple","green"))
14
15        plot(dataset("datasets", "CO2"), x=:Conc, y=:Uptake,
16    group=:Plant, linestyle=:Treatment, color=:Type, Geom.line,
17    Scale.linestyle_discrete(order=[2,1]))
```

Also used for scaling of axis:

- Scale.x_continous(format=, minvalue,maxvalue)
- Scale.x_discrete
- Scale.xlog10 : Added as elements to change entire plot
- Scale.ylog10

Listing 20: Scale examples

```
1
2         p2 = plot(mammals, x=:Body, y=:Brain, label=:Mammal,
3                 Geom.point, Geom.label,
4                 Scale.x_log10, Scale.y_log10)
5
6         p3= plot(Diamonds, x=:Price, y=:Carat,
7                 Geom.histogram2d(xbincount=25, ybincount=25),
8         Scale.x_continuous(format=:engineering))
```

9.2.4 Guides

Types of Guides:

- Annotation (inc shapes to highlight points)
- Keys : colour, shape or size
- Title labels
- xrug,yrug : show distribution of points on axis
- ticks : show plot lines

Listing 21: Guide examples

```

    pb = plot(iris, x=:SepalLength, y=:PetalLength, color=:Species,
            Geom.point,
    Guide.colorkey(title="Iris", pos=[0.05w,-0.28h]) )

    plot(dataset("ggplot2", "diamonds"), x="Price", Geom.histogram,
            Guide.title("Diamond Price Distribution"))
```

9.2.5 Stats

Can also uses in built stat transformations ie:

- Density
- Quantile
- Doge: Grouped and shown value

9.3 Compositing

Types:

- Facets: Share same axis dims but seperate
- Stack: Diff datasets and axis but together
- Layers: Single plot

9.3.1 Layers

Listing 22: Layer examples

```

    l = layer(data,elements;mapping)
    # Can not include Scale,coordinates or guides
    #use plot for that

    plot(layer1,layer2..)

    plot(data
            layer(elements;mapping)
            layer(...))
```

10 Profiling

Julia has built in profiling, allowing code lines to be timed to discover code bottlenecks.

Checks how often lines appear in a set of backtraces, the line coverage is therefore dependant on the backtrace sampling freq.

Pros: No need to change code to measure and little computational overhead.

10.0.1 Example

Listing 23: Profiling

```
1
2       using Profile
3
4       @profile myfunc()
5
6       Profile.print() #Output to stderr backtraces
7
8       #Run test 100 times
9
10      @profile for( i=1:100; myfunc();end)
11
12      Profile.clear() #Clear buffer
13
14      @time func() #Measure how long to run func
```

The number of backtraces shows how expensive that line is.

10.0.2 Memory allocation

Use @time or @allocation to check the memory allocation. Can also check the cost of garbage collection.

To check line by line allocation, use flag trackallocation. Remove compiler overhead before measuring allocation by running Profile.clear_malloc_data() after executing commands.

GC.enable_logging(true) set to true to check collection cost.

11 Performance tips

Performance critical code should be inside a function.

AVOID:

- Untyped global variables, if they must be used annotate the type at point of use.
- Problems with type stability
- Many temporary small arrays

At point of use annotation:

```
for i in x :: Vector{Float64}
```

11.1 Wrappers

Better to define the type of the wrapper than only the its elements. As you dont want the high level struct type to be ambiguous:

Listing 24: Structs

```
1
2       mutable struct Ty{T<: AbstractFloat}
3               a::T
4       end
5
6       #This is better than
7
8       mutable struct Ty
9               a::Float64 #should be concrete type
10      end
```

Functions using the same struct but different types of elements should be explicit of the type of elements

Listing 25: functions

```
1
2       function func(c::Ty{<:AbstractArray{<:Integer}})
3
4       #If a type in vector is known should declare it!
5       function func(a::Vector{Any,1})
6               x = a[1]::Int32
```

However, abstract type annotation for values and elements can hinder performance.

Dont use compound functions, ie same function used differently for types use multiple dispatch.

11.2 How should functions be returned?

A function should always return the same type, so if the type is ambiguous convert before return.

```
zero(x) #Gives a 0 val in type of x
oftype(x,y) #Gives y as type of x
```

Variables changing type within the function should also be avoided.

11.3 How should functions be broken down?

Generally functions of made of the setup and compute on the setup vars. The compute should broken into its own function.

11.4 Types used as parameters for other types

If a var is created using an unknown type ie a matrix or array then its type will be checked on every access.

```
fill(5,ntuple(d->3,N)) #If N is unknown then will create an unknown type matrix.
::Val{N} #is used a concrete type to specify dimensions, pass as Val(N) in func
```

11.5 What is dangerous when using Multi Dispatch?

If at compile time Julia does not know the type it will be checked at run time slowing down compute.

Listing 26: Multi Dispatch

```
1
2       struct Car{:make,:model}
3               year::Int
4       end
5
6       Array{Car{:Honda,:Accord},N}
7       #This is contiguous and known
8
9       ar = [Car{:Honda,:Accord}(2),Car{:Honda,:Buggy}(3)]
10      #Non contiguous will be determined at runtime
```

The above situation would be bad if used as method of multi dispatch as many lookups on each type as the array is iterated will be done. Will also take a lot more compile time memory as all versions of functions will need to be compiled for each type ie for push!.

11.6 What are the array types used in Julia?

Arrays are column major, the first index changes most rapidly (row). Meaning the inner most loop should be iterated on rows, the outer loop on cols. Cols are faster to get than rows.

11.7 Should outputs for func allocated before?

The output of a function should be preallocated with a defined type and the function should be used to change its values.

11.8 How should nested dot calls be used?

Nested dot calls are fusing, they are combined at the syntax level into a single loop. And use preallocated arrays instead of temp ones (SHOULD BE USED WHENEVER POSSIBLE!).

Listing 27: Nested dot calls

```
1
2       f(x) = 3x.^2 + 4x + 7x.^3
```

```
3         fdot(x) = @. 3x^2 + 4x + 7x^3
4
5         # @. converts all vector ops into a dot operation
6         # fdot(x) is 10x faster
```

11.9 How should views be used?

A slice creates a copy of the array, which is okay if many operations are to be done on the copy, however, could be worse if the cost of the copy outweighs the operations to be done on it.

Can instead use @view to reference a subarray of the original array.

Also contiguous are better and if possible arrays should be converted into a contiguous array before operation.

11.10 Other

- Static array pkg if array is to be fixed size
- Avoid string interpolation for I/O (dont use $a)
- Fix all warnings
- Avoid unnecessary arrays
- Use fld(x,y) and cld(x,y) instead of floor and ceil
- Annotations available to reduce Julia functions for sake of speed ie don't check boundaries.
- Use @code_warntype to show non concrete types in red.