Julia Cheat Sheet

Mo D Jabeen

October 18, 2022

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

    newstruct = name(...)

    # Internal constructors are used to place constraints on the code

    mutable struct name
        meh::AbstractString
        numb::Int

        name(blah::AbstractString)= new(meh,4)
        # this enforces if a struct without
        #a number is given 4 is placed
    end
```

Listing 5: Tenancy operations

```

    x > 0 ? 1 : -1
    # 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

```

    struct name{T<:Integer} <: Real
    # <: shows all values are included in that set
```

```
    # {for arg} outer for object

        num::T
        den::T #ensures both are of type T

        #Function checks if the input numbers are empty for every object
        function name{T}(num::T, den::T) where T <: Integer

            if num == 0 && den == 0
                error("invalid")
            end
            new(num, den)
        end
    end


    name(n::Int, d::Float) = name(promote(n,d)...)
    #Outer constructor
    #Promote converts values of a single type to the same type
    #choosing the type to work with both


    # MULTI-DISPATCH FUNCTION

    function blah(n::Int, d::Int) = println('meh')

    function blah(x::Int, y::name) = println(x*y.num)
    #This func now has two methods (multi dispatch)
```

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

```

macro name()

end

@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

```

struct MyNumber
x::Float64
end
# output

for op = (:sin, :cos, :tan, :log, :exp)
@eval Base.$op(a::MyNumber) = MyNumber($op(a.x))
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

```

t = @task func()
OR
t= @task begin ... end
OR
t = Task(func)

schedule(t,[val],error) # Allocate task to scheduler, pass val

# if error true, val passed as an exception

@async func() same as schedule(@task func())

asyncmap(func, collection ,ntask,batch)

# Return collection with the func executed on by ntasks.
# Batch executes on collection in groups set by number of batch.

yield() #Switch to scheduler to allow another task to run
yield(t) #Switch to task t.

Condition() #Edge triggered event source
thread.condition - thread safe version

Event() #Level triggered event source

notify(condition,val,all,error) #Wake up tasks waiting for condition

semaphore(sem_size) #counting with max at semsize

acquire(s) #get a semaphore, blocks if none available
release(s)

#Use below format for locking
```

```
lock()
try
...
finally
unlock()
end

bind(channel, task)
```

Listing 10: Async wait functions

```
wait([x])

x {
        Channel: Wait for val
        Condition: Wait for notify
        Process: wait for process to exit
        Task: wait for task to finish
        RawFD: change the file descriptor
}

#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

```
c = Channel{Type}(limit) #limit is max number of objects in queue

put!(channel,data) # Place data into channel
take!(channel) #Read data from channel

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

```
Threads.@threads [schedule] for ... end

[schedule] {
        default: :dyanmic assumes equal load per thread, cant
        guarantee thread id on an iteration
        :static one task for thread, can guarantee same id for an
        iteration
}

threads.foreach(f,c,ntasks) #operate function on channel with
n threads

Threads.@spawn func() #Create task and schedule to run on any
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

```
@debug #Auto set to not output to stderr
@info # mid level
@warn #Higher level
@error #highest level, generally not needed
# (use exceptions instead)

@__ msg var x=var y=func() #msg in markdown

@__ "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

```
ConsoleLogger([steam,] min_level=Info) #stream can be a file io
SimpleLogger([stream,] min_level=Info)

global_logger(logger)
with_logger(logger) do ... end #Local logger
```

Listing 15: Log filter

```
disable_logging(level) #disable below this level
```

```
should_log(logger, level) #return true if accepts this level

handle_message(logger, level, message, val ...)
```

Listing 16: Example log

```

# Open a textfile for writing
io = open("log.txt", "w+")
IOStream(<file log.txt>)

# Create a simple logger
logger = SimpleLogger(io)
SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())

# Log a task-specific message
with_logger(logger) do
        @info("a context specific log message")
    end

# Write all buffered messages to the file
flush(io)

# Set the global logger to logger
global_logger(logger)
SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())

# This message will now also be written to the file
@info("a global log message")

# Close the file
close(io)

# Create a ConsoleLogger that prints any log
#messages with level >= Debug to stderr
debuglogger = ConsoleLogger(stderr, Logging.Debug)

# Enable debuglogger for a task
with_logger(debuglogger) do
        @debug "a context specific log message"
    end

# Set the global logger
global_logger(debuglogger)
```