Julia Cheat Sheet

Mo D Jabeen

October 7, 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
```

```
6
7       a = map((x) -> x^2, [1,3,5,3]) # map performs func on each array element
8
9       foreach(func, collection) #operate func on each val of collection
```

Listing 4: Struct definition.

```
1
2       mutable struct name
3               string::AbstractString
4               boolean::Bool
5               age::Int
6               a::Array{Int,5}
7       end
8
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)
```

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 guarantee thread id
6               :static one task for thread, can guarantee same id for an iteration
7       }
8
9       threads.foreach(f,c,ntasks) #operate function on channel with n threads
10
11      Threads.@spawn func() #Create task and schedule to run on any available thread
```