Julia Cheat Sheet

Mo D Jabeen

August 16, 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
```

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
```