More about mutable/unmutable, variable bindings

A variable in Julia is bound to (refers to, points to) a value

\[ \text{var} \rightarrow \text{value} \]

- var is a memory address
- value is stored at that address

- var2 = var means that var2 will point to the same value as var

when an unmutable object is changed (e.g., var=var+1)
- ‘value’ may not change, but var points to another address with new value

when a mutable object changes
- the address does not change but the contents of that address change

An array is an example of a mutable object
- the binding is to the first memory address where the array is stored

Of relevance to how arguments are passed (from Julia doc):

Julia function arguments follow a convention sometimes called "pass-by-sharing", which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable bindings (new locations that can refer to values), but the values they refer to are identical to the passed values. Modifications to mutable values (such as an array) made within a function will be visible to the caller.
More about functions

function func(a,b,c)
  . . .
  return d,e
end

without return, the last evaluated expression is returned

return or return nothing returns object ‘nothing’

Single-expression function

func(arguments) = expression
func(a,b,c) = a+b–c

expression can be multiple statements between begin ... end

Functions are objects that can be assigned, passed to other functions, etc

func2=func

somefunction(func,...)

Anonymous function

Example from Julia documentation

julia> map(x -> x^2 + 2x - 1, [1, 3, -1])
3-element Vector{Int64}:
  2
  14
  -2

map(function,collection) is a Base function, performs function on each element of collection

Read about: optional arguments, Varargs (arbitrary number of arguments), keywords...
**Modules**
Can be used to organize codes
- make modules with functions and data structures for specific tasks
- variables and functions can be exported to code block using the module

```
module ModName
...
export vari1, func1
...
end
```

using .ModName

if module declared in same file

```
include("modname.jl")
```

if in a different file
- include() inserts the contents of the file

Even functions/data not exported can be accessed:  `ModName.vari2`  `Modname.func2`
Those exported do not need `ModName`

. before module name required if the module is not installed as a package
- only do this if you have developed a stable module

Example in module.jl, to be used with main.jl
The “let” block; a simple way to store values in functions

We often want to store the “internal state” of some function without having to pass that state as an argument.

For example, rand() can be called without any argument - but clearly there must be some internal state that is somehow saved.

References to data (pointers) can be permanently saved in “let” blocks - functions defined inside a let block can access these pointers.

Example, part of letblock.jl (random number generator, inside a module)

```julia
let
    r = Ref(convert(UInt64,1))
    global function ran64()
        r[] = r[] * a + c
    end
end
```

r is a reference to an unsigned integer
the value at r is accessed by r[]
- would be r[i] for element i of a 1d array

Why not just use r declared in the global scope?
- for efficiency, avoid using global variables

The function must be declared global to make it accessible outside let-end
- global function objects are treated as constants, not slowing things down
- the integers a and c are declared as constants before let

The let block is a local hard scope, many other uses (see Julia doc)