More about mutable/unmutable, variable bindings

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

var → 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

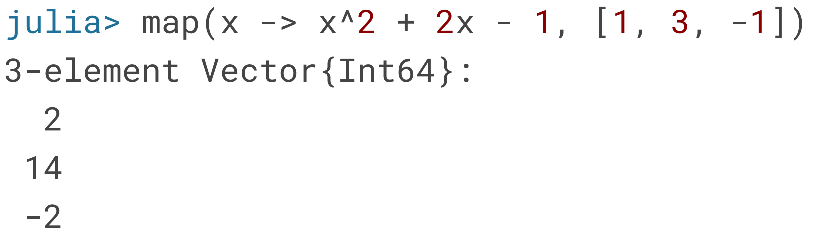
Functions are objects that can be assigned, passed to other functions, etc

`func2=func` `somefunction(func,…)`

Read about: optional arguments, Varargs (arbitrary number of arguments), keywords…

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

Modules

Can be used to organize codes

- make modules with functions and data structures for specifc 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

Even functions/data not exported can be accessed:

```
ModName.vari2
Modname.func2
```

```
include(“modname.jl”)
using .ModName
```

if in a different file

- include() inserts the contents of the file

Those exported do not need ModName

. before module name required if the module is not installed as a package

- only make a package if you have developed a stable module

Example in module.jl, to be used with main.jl

Using modules available in the “community”

Packages (which may involve several modules) that are registered can be added with the REPL package manager

Information about the registry and all its packages available here

https://github.com/JuliaRegistries/General

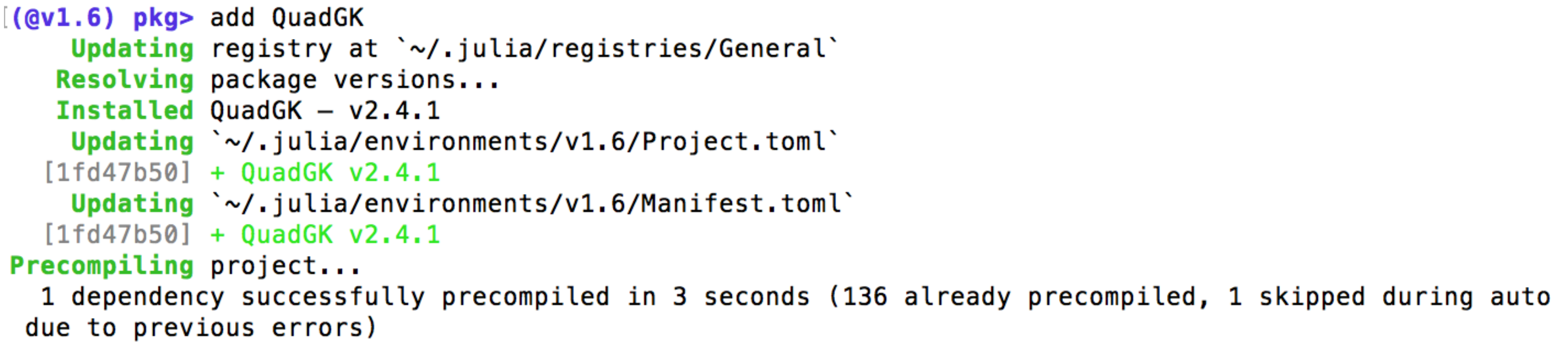
You can register your own package if you make something useful!

There is a search function, but just googling “Julia whatyouwant” may be better

Example: after googling “Julia integration” I quickly found QuadGK

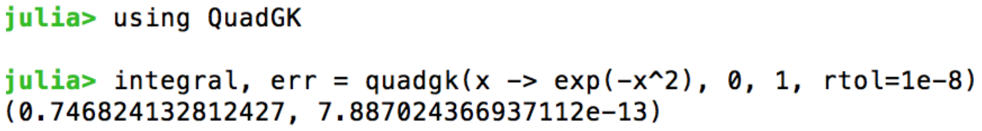
https://juliapackages.com/p/quadgk

Installation in the REPL package manager (“]” at the Julia prompt)



```
[(@v1.6) pkg> add QuadGK
    Updating registry at `~/.julia/registries/General`
   Resolving package versions...
   Installed QuadGK ─ v2.4.1
    Updating `~/.julia/environments/v1.6/Project.toml`
  [1fd47b50] + QuadGK v2.4.1
    Updating `~/.julia/environments/v1.6/Manifest.toml`
  [1fd47b50] + QuadGK v2.4.1
Precompiling project...
  1 dependency successfully precompiled in 3 seconds (136 already precompiled, 1 skipped during auto
 due to previous errors)
```

Now we can integrate functions of one variable:



```
julia> using QuadGK

julia> integral, err = quadgk(x -> exp(-x^2), 0, 1, rtol=1e-8)
(0.746824132812427, 7.887024366937112e-13)
```