Introducing Julia/Modules and packages

1 Modules and packages

Julia code is organized into files, modules, and packages. Files containing Julia code use the .jl file extension.

1.1 Modules

Related functions and other definitions can be grouped together and stored in **modules**. The structure of a module is like this:

module MyModule end

and in between these lines you can put functions, type definitions, constants, and so on.

1.1.1 Installing modules

To use an official Julia module on your own machine, you download and install the package containing the module from the main GitHub site. There's a list of packages available at pkg.julialang.org. To download and install one, you give its name to the Pkg.add() function:

julia> Pkg.add("Calculus") INFO: Installing Calculus v0.1.6 INFO: Package database updated julia>

(If you are not directly connected to the internet, you need to give the name of a proxy before calling the package installer.)

To find out where packages are installed, type

julia> Pkg.Dir.path() "/my_home_directory/.julia/v0.4"

1.1.2 Using modules

After installation, to start using functions and definitions from the module, you tell Julia to make the code available to your current workspace, with the using statement, which accepts the names of one or more installed modules:

julia> using Calculus julia>

Now, all the definitions in the Calculus module are available for use. If the definitions inside Calculus were exported by the module's author, you can use them without the module name as prefix (because we used using):

julia> derivative(sin, pi/2) 0.0

If the package author(s) don't export the definitions, or if we use import rather than using (See below), you can still access them, but you have to type the module name as a prefix:

julia> Calculus.derivative(sin, pi/2) 0.0

but that's unnecessary in this example, as we've seen.

When you write your own modules, the functions that you choose to export can be used **without** the module name as prefix, Those that you don't export can still be used, but only if they are prefixed with the module name.

For example, in the module called MyCoolModule, the mycoolfunction() was exported. So the prefix is optional:

julia>using MyCoolModule julia> MyCoolModule.mycoolfunction() "this is my cool function" julia> mycoolfunction() "this is my cool function"

Inside the module, this function was exported, using the export statement:

module MyCoolModule export mycoolfunction function mycoolfunction() println("this is my cool function") end end

1.1.3 using and import

import is similar to using, but differs in a few ways, for example, in how you access the functions inside the module. Here's a module with two functions, one of which is exported:

module MyModule export mycoolfunction function mycoolfunction() println("this is my cool function") end function mysecretfunction() println("this is my secret function") end end

Import the module:

julia> import MyModule julia> mycoolfunction() ERROR: mycoolfunction not defined julia> MyModule.mycoolfunction() "this is my cool function"

Notice that mycoolfunction() could be accessed **only** with the module prefix. This is because the MyModule module was loaded using **import** rather than **using**. Similarly for mysecretfunction():

julia> mysecretfunction() ERROR: mysecretfunction not defined julia> MyModule.mysecretfunction() this is my secret function

Unlike using, import doesn't let you refer to a number of modules on the same line:

using Color, Calculus, Cairo

Another important difference is when you want to modify or extend a function from another module. You can't use using, you have to import the specific function.

1.1.4 Include, require, reload

If you want to use code from other files that aren't contained in modules, there are the following functions:

include(pathname) evaluates the contents of the file in the context of the current module, searching relative to the path of the source file from which it is called. This is useful for building code from a number of smaller files.

require(filename) loads the file in the context of the Main module. It first looks in the current working directory, then looks for package code under the current package directory (as reported by Pkg.dir()), then tries paths stored in the global array LOAD_PATH.

reload(filename) is like require, but forces the file to be loaded again. You would probably use this while developing your code.

1.2 How does Julia find a module?

Julia looks for module files in directories defined in the LOAD_PATH variable.

julia> LOAD_PATH 2-element Array{Union(ASCIIString,UTF8String),1}: "/Applications/Julia-0.3.0.app/Contents/Resources/julia/local/share/julia/site/v0.3" "/Applications/Julia-0.3.0.app/Contents/Resources/julia/share/julia/site/v0.3"

To make it look in other places, add some more using push!:

julia> push!(LOAD_PATH, "/Users/me/julia") 3-element Array{Union(ASCIIString,UTF8String),1}: "/Applications/Julia-0.3.0.app/Contents/Resources/julia/local/share/julia/site/v0.3" "/Applications/Julia-0.3.0.app/Contents/Resources/julia/share/julia/site/v0.3" "/Users/me/julia"

And, since you don't want to do this every time you run Julia, put this line into the file ~/.juliarc.jl.

Julia looks for files in those directories in the form of a package with the structure

ModuleName/src/file.jl

Or if not in Package form (see below), it will look for a filename that matches the name of your module:

using MyModule

Would look in the LOAD_PATH for a file called MyModule.jl and load the module contained in that file.

1.3 Packages

The built in module Pkg provides a number of functions for managing the packages you've installed. We've seen Pkg.add():

julia> Pkg.add(packagename) julia> using Calculus julia> derivative(cos, 1.0) −0.8414709847974693

1.3.1 Current status of packages

The following functions in the Pkg package are very useful, particularly at present when packages are constantly being updated to keep pace with Julia's rapid development.

Pkg.installed() returns the version numbers of your installed packages, in a form that you can examine with Julia code:

julia> Pkg.installed() Dict{ASCIIString,VersionNumber} with 73 entries: "Coverage" => v"0.0.4" "ASCIIPlots" => v"0.0.2" "Lazy" => v"0.6.0" "ImmutableArrays" => v"0.0.6" "Luxor" => v"0.0.0-" "Tk" => v"0.2.16" "ZMQ" => v"0.1.15" "ArrayViews" => v"0.4.8" "DataStructures" => v"0.3.5" "Compat" => v"0.2.2" "LNR" => v"0.0.1" "Calculus" => v"0.1.5" "Elliptic" => v"0.1.1" "GZip" => v"0.2.13" "HTTPClient" => v"0.1.4" "Lint" => v"0.1.51" "Cairo" => v"0.2.20" "HttpParser" => v"0.0.10" "DataFrames" => v"0.5.12" "Requests" => v"0.0.6" "Showoff" => v"0.0.2" "SQLite" => v"0.1.6" "Distributions" => v"0.6.1" "ICU" => v"0.4.4" "ZipFile" => v"0.2.2" "FactCheck" => v"0.2.2" "FixedPointNumbers" => v"0.0.4" "GnuTLS" => v"0.0.1" "SHA" => v"0.0.3" "Plotly" => v"0.0.0-" "TexExtensions" => v"0.0.2" "KernelDensity" => v"0.0.2" "Iterators" => v"0.1.7" "Gadfly" => v"0.3.9" "Dates" => v"0.3.2" "Contour" => v"0.0.6" "Jewel" => v"0.9.1"

Pkg.status() shows additional information about your installed packages.

julia> Pkg.status() 25 required packages: - ASCIIPlots 0.0.2 - Cairo 0.2.21 - Calculus 0.1.5 - Calendar 0.4.2 - Color 0.3.15 - Compose 0.3.10 master - Coverage 0.0.6 - Dates 0.3.2 - Distances 0.2.0 - Gadfly 0.3.9 - IJulia 0.1.16 - Images 0.4.26 - JSON 0.4.0 - Jewel 1.0.4 - LightXML 0.1.9 - Lint 0.1.61 - Logging 0.0.4 - PyPlot 1.5.0 - RDatasets 0.1.1 - Requests 0.0.6 - SIUnits 0.0.2 - SQLite 0.2.0 - TestImages 0.0.7 - Tk 0.2.16 - Winston 0.11.7 53 additional packages: - ArrayViews 0.4.8 - BinDeps 0.3.7 - Codecs 0.1.3 - Compat 0.2.9 - Contour 0.0.6 - DataArrays 0.2.8 - DataFrames 0.6.0 - DataStructures 0.3.5 - Distributions 0.6.3 - DualNumbers 0.1.1 - Elliptic 0.1.1 - FactCheck 0.2.5 - FixedPointNumbers 0.0.6 - GZip 0.2.13 - Geometry2D 0.0.0- master (unregistered) - GnuTLS 0.0.1 - Grid 0.3.7 - HTTPClient 0.1.4 - Hexagons 0.0.2 - Homebrew 0.1.13 - HttpCommon 0.0.11 - HttpParser 0.0.10 - ICU 0.4.4 - ImmutableArrays 0.0.6 - IniFile 0.2.4 - Iterators 0.1.7 - Ju-

liaParser 0.5.3 - KernelDensity 0.1.0 - LNR 0.0.1 - LaTeXStrings 0.1.2 - Lazy 0.8.3 - LibCURL 0.1.4 - Loess 0.0.3 - Luxor 0.0.0- master (unregistered) - NaNMath 0.0.2 - Nettle 0.1.7 - Optim 0.4.0 - PDMats 0.3.1 - PhilipsHue 0.0.0- master (unregistered) - Plotly 0.0.0- master (unregistered) - PyCall 0.7.3 - REPLCompletions 0.0.3 - Reexport 0.0.2 - Requires 0.1.1 - SHA 0.0.3 - Showoff 0.0.3 - SortingAlgorithms 0.0.2 - StatsBase 0.6.10 - TexExtensions 0.0.2 - URIParser 0.0.3 - ZMQ 0.1.15 - ZipFile 0.2.3 - Zlib 0.1.7

To update any packages that are out of date compared with the official master versions stored on GitHub, use Pkg.update():

julia> Pkg.update() INFO: Updating METADATA... INFO: Upgrading DataStreams: v0.0.4 => v0.0.5 INFO: Upgrading NullableArrays: v0.0.2 => v0.0.3 INFO: Upgrading SQLite: v0.3.1 => v0.3.2 julia>

1.3.2 Structure of a package

Julia uses git for organizing and controlling packages. By convention, all packages are stored in git repositories, with a ".jl" suffix. So the Calculus package is stored in a Git repository called Calculus.jl. This is how the Calculus package is organized in terms of files on disk:

Calculus.jl/ # this is the main package directory for the Calculus package src/ # this is the subdirectory containing the source Calculus.jl # this is the main file - notice the capital letter module Calculus # inside this file, declare the module name import Base.ctranspose # and import other packages export derivative, check_gradient, # export some of the functions defined in this package ... include("derivative.jl") # include the contents of other files in the module include("check_derivative.jl") include("integrate.jl") end # end of Calculus.jl file derivative.jl # this file contains code for working with derivatives, function derivative() # and is included by Calculus.jl ... end ... check_derivative.jl # this file concentrates on derivatives, function check_derivative(f::...) # and is included by "include("check_derivative.jl")" in Calculus.jl ... end ... integrate.jl # this file concentrates on integration, function adaptive_simpsons_inner(f::Function..... # and is included by Calculus.jl ... end ... symbolic.jl # concentrates on symbolic matters; included by Calculus.jl export processExpr, BasicVariable, ... # these functions are available to users of the module import Base.show, ... # some Base functions are imported, type BasicVariable <: AbstractVariable # so that more methods can be added to them ... end function process(x::Expr) ... end ... test/ # this directory contains the tests for the Calculus module runtests.jl # this file runs the tests using Calculus # obviously the tests use the Calculus module... using Base.test # and the Base.test module... tests = ["finite_differerence", ... # the test file names are stored as strings... for t in tests include("$(t).jl") # ... so that they can be evaluated in a loop end ... finite_difference.jl # this file contains tests for finite differences, @test ... # its name is included and run by runtests.jl ...

2 Global variables and scope

It's common to want to control access to variables. Sometimes you want to define a variable so that every function in the module can use the same value. To do this, you can use the **global** keyword when defining the variable.

Here's an example of a module called Sundial, and how you can access the variable called latitude, which presumably will be useful for a number of different functions inside the module.

module Sundial global latitude = 52 export latitude, get_lat, set_lat, use_lat function get_lat() println("in Sundial.get_lat, latitude is $latitude") end function set_lat(lat) global latitude latitude = lat println("in Sundial.set_lat, latitude is now $latitude") end function use_lat_fail() try println("in Sundial.use_lat_fail, latitude is $latitude") catch println("couldn't find it") end latitude = 0 println("in Sundial.use_lat_fail, latitude was set to 0, is $latitude") end function use_lat() global latitude try println("in Sundial.use_lat, latitude is $latitude") catch println("couldn't find it") end latitude = 0 println("in Sundial.use_lat, latitude was set to 0, is $latitude") end end

The second line of the module uses the **global** keyword and sets the value of the variable to 52. Other functions in the module can access this variable if they use the **global** keyword. These functions, and the name of the variable, are exported using this line:

export latitude, get_lat, set_lat, use_lat

Before you load the module into Julia, there is no variable called latitude:

julia> latitude ERROR: latitude not defined

After adding the module:

julia> using Sundial

the value of latitude is available, and can be used without the prefix, because the module exported it and because we loaded the module with using:

julia> latitude 52

It's also available with its module prefix:

julia> Sundial.latitude 52

but we can't change its value, because we're not "in" the Sundial module, as the error message tells us:

julia> Sundial.latitude = 2 ERROR: cannot assign variables in other modules

Our get_lat() function can access the variable's value:

julia> Sundial.get_lat() in Sundial.get_lat, latitude is 52

Although you can read the value easily, you can't set it.

For example:

julia> latitude = 40 Warning: imported binding for latitude overwritten in module Main 40

The warning here tells you that there are now two variables called latitude, a new one you've created in the Main context, and the existing one in the Sundial context. They have different values:

julia> latitude 40 julia> Sundial.latitude 52 julia> Main.latitude 40

The get_lat() and set_lat() functions provided by the module allow you to access and change the values from anywhere:

julia> Sundial.get_lat() in Sundial.get_lat, latitude is 52 julia> Sundial.set_lat(45) in Sundial.set_lat, latitude is now 45 julia> Sundial.get_lat() in Sundial.get_lat, latitude is 45

The set_lat() function used the **global** keyword at the start of the definition. This means that code inside the function definition can change the value.

Without the **global** keyword, as in the use_lat_fail() function, any attempt to change the module's latitude variable fails.

julia> Sundial.use_lat_fail() couldn't find it in Sundial.use_lat_fail, latitude was set to 0, is 0

But wait: the use_lat_fail() function **appears** to work - it says that latitude was set to 0. This is a new, local, latitude symbol created just for the duration of the user_lat_fail() function. However, you can quickly check to reassure yourself that the "real" module's variable has not been set to 0:

julia> Sundial.get_lat() in Sundial.get_lat, latitude is 45

To change the variable, the use_lat() (without the '_fail') function can set the module's latitude variable to 0, thanks to the use of the **global** keyword:

julia> Sundial.use_lat() in Sundial.use_lat, latitude is 45 in Sundial.use_lat, latitude was set to 0, is 0

The warning you received above, about overriding the imported binding in Main, prepares you for the following:

julia> latitude 40 julia> Sundial.latitude 0 julia> Main.latitude 40 julia> Sundial.get_lat() in Sundial.get_lat, latitude is 0

The Main context's version of latitude, also available without the Main prefix, is still 40; the

use_lat()

function changed the module's latitude variable to 0. You can't now access the module's version of latitude without using the Sundial prefix.

The correct way to set the value of the module's global variable is to use set_lat():

julia> Sundial.set_lat(45) in Sundial.set_lat, latitude is now 45 julia> latitude 60 julia> Sundial.latitude 45 julia> Main.latitude 60 julia> Sundial.get_lat() in Sundial.get_lat, latitude is 45

The 'wrong' ones are still 60, but the module's version is correctly set to 45.

3 Text and image sources, contributors, and licenses

3.1 Text

- **Introducing Julia/Modules and packages** *Source:* https://en.wikibooks.org/wiki/Introducing_Julia/Modules_and_packages?oldid=3098300 *Contributors:* Cormullion, UncleOxidant, Fapae and Anonymous: 3

3.2 Images

3.3 Content license