Modules and packages

Solution | Agent-based modelling, Konstanz, 2024

Henri Kauhanen

7 May 2024

1. `using` vs. `import`

Imagine the following scenario: Alice writes some Julia code which defines a function `do_cool_stuff()` (which does cool stuff), and Bob also writes some Julia code that also defines a function named `do_cool_stuff()`, which also does cool stuff but some slightly different cool stuff than Alice's function. In other words, the functions have the same name but do slightly different things.

Now imagine Carlos has access to both Alice's and Bob's code. If Carlos executes the function `do_cool_stuff()`, what is going to happen? Will Alice's code run, or will Bob's code run?

The standard solution to this kind of problem is to put code in **modules**. A module defines a **namespace**. This makes it possible to use the same function names (as well as names for custom types etc.) in different modules.

If Alice has created a module called `CodeOfAlice` and Bob's module is called `CodeOfBob`, then the following commands can be used to distinguish between the two different but homonymous functions:

```
CodeOfAlice.do_cool_stuff()
CodeOfBob.do_cool_stuff()
```

Now we can finally explain the difference between `using` and `import`. Both commands can be used to load a module into memory. The difference is:

- `using` allows the user to refer to a function such as `do_cool_stuff()` without specifying the module the function comes from. This can cause conflicts, if more than one module is used that defines the same name. In that case, Julia will throw a warning, and it is up to you (or Carlos, in this case) to make sure that you're not accidentally calling the wrong function!

- `import` does not allow the above behaviour. With `import`, you **must** specify the module name every time you call a function.

In summary, you can do either

```
using CodeOfAlice
do_cool_stuff()
```

or

```
import CodeOfAlice
CodeOfAlice.do_cool_stuff()
```

2. `include()`

The command `include()` is used when you want to import into your session Julia code which lives in an external file (typically, with the `.jl` file extension) but which is not necessarily organized into a module. We used this before to load the plotting code I had written for the bird agents.

3. `export`

The command `export` defines which functions and objects defined inside a module will be visible to outside users. For example, in the above example, both Alice and Bob have included the line `export do_cool_stuff` in their code, to make the function available to outside users of the module.

Functions which are not `export`ed can still be used. However, in that case, the module name must always be specified, regardless of whether the module has been loaded using `using` or `import`. In other words, non-`export`ed stuff can only be accessed like this: `ModuleName.function_name()`.

4. The `VariationalLearning` module

Here, I am following the logic of the lecture on Speaking and listening, omitting the older `LearningEnvironment` type which we will no longer need. I've decided to `export` every type and function defined by the module. In this case it makes sense, since the external user may conceivably want to make use of all of them. (We will later see examples of so-called "helper" functions which need not be exposed to end-users.)

```
module VariationalLearning

# we need this package for the sample() function
using StatsBase

# we export the following types and functions
export VariationalLearner
export speak
export learn!
export interact!

# variational learner type
mutable struct VariationalLearner
  p::Float64      # prob. of using G1
  gamma::Float64  # learning rate
  P1::Float64     # prob. of L1 \ L2
  P2::Float64     # prob. of L2 \ L1
end

# makes variational learner x utter a string
function speak(x::VariationalLearner)
  g = sample(["G1", "G2"], Weights([x.p, 1 - x.p]))

  if g == "G1"
    return sample(["S1", "S12"], Weights([x.P1, 1 - x.P1]))
  else
    return sample(["S2", "S12"], Weights([x.P2, 1 - x.P2]))
  end
end

# makes variational learner x learn from input string s
function learn!(x::VariationalLearner, s::String)
  g = sample(["G1", "G2"], Weights([x.p, 1 - x.p]))

  if g == "G1" && s == "S1"
    x.p = x.p + x.gamma * (1 - x.p)
  elseif g == "G1" && s == "S2"
    x.p = x.p - x.gamma * x.p
  elseif g == "G2" && s == "S2"
    x.p = x.p - x.gamma * x.p
  elseif g == "G2" && s == "S1"
    x.p = x.p + x.gamma * (1 - x.p)
```

```
  end

  return x.p
end

# makes two variational learners interact, with one speaking
# and the other one learning
function interact!(x::VariationalLearner, y::VariationalLearner)
  s = speak(x)
  learn!(y, s)
end

end   # this closes the module
```

It now makes sense to save this code in a file. I've saved it in `VariationalLearning.jl`. This way, we can easily **reuse** our code in the future and not have to rewrite it again and again.

Bonus

Okay, so now we know what a module is. What about the other thing mentioned in the title, packages? What is a package?

Essentially, a package is just a module which is being maintained using a version control system and is offered to the rest of the world, usually (but not always) through the official Julia registry. (These are the things you install with `Pkg.add("PackageName")`.)