Week 4 Walkthrough: Packages

2025-01-24

Packages

Hello again! This week we will be talking all about **Packages**, one of the best things about `R`! If you've ever heard about Free and Open Source Software (FOSS), know that `R` is an example of such software. If you *haven't*, then know that FOSS means all the content available and the entire program itself was coded up by members of the community just like you (but with a few more years of experience)! This means anyone anywhere can contribute to `R` and add more functionality, which helps `R` stay on the cutting edge of statistical methods. A lot of these more niche contributions end up as **Packages**, which we will discuss in detail!

What is a Package?

A **package** is a collection of `R` **objects** and **functions**. Inside of every **package** there will generally be a mix of new functions to add *even more* flexibility to `R` or give you access to public datasets.

The Package `base`

By default, `R` loads a few packages, the most important of which is `base`. The `base` package contains every function we have looked at so far and *a lot* more! From the `base` **package** alone, you could (eventually) write your own code to recreate almost any other **package**! So a good handle on `base` can get you pretty far, to say the least.

The *Philosophy* of Public Packages

I care *a lot* about open-source software and responsible conduct of research. If you don't, skip this section!

While `base` is super strong and flexible itself, other **packages** offer convenient ways to do more. People publish **packages** alongside academic articles all the time to make it easy to reproduce their work, a big part of responsible conduct of research!!! And if people publish **packages** to make it easy on us, why would be want to reinvent the wheel and rewrite their code?

Coders are lazy creatures, but also generous ones! The sharing of **packages** is just one of many ways coders can contribute to the broader community and help save everyone a little effort.

Using Packages

In the eternal words of Andre 3000: "Y'all don't wanna hear me, you just wanna dance." So let's get down to brass tacks and see how we would actually put any of this into practice.

Everything we've done so far has been in the `base` package of `R` (default `R`), but for much of the rest of the course we will be using the `TidyVerse`, a collection of **packages** developed explicitly with your learning in mind! They redefine the "grammar" (so to speak) of a lot of `R` so that it feels a little bit more natural for a lot of people. Don't worry though, everything you've learned so far still translates!

If you installed `R` to your personal computer, whenever you want to use a new **package** you will first have to run the line `install.packages("...")`, where the dots represent the **package**'s (case sensitive!) name. In the example of `dplyr`, we would run the below. Note the quotation marks around `dplyr`! This is very

important when installing **packages** as before a **package** is installed, it doesn't exist on your computer. We put it in quotation marks `" "` then so that `R` can search for the name in an online repository.

```
install.packages("dplyr")
```

If you're using the web version of `R`, then you don't need to worry about this! It already has all of the **packages** you'd want installed! Then we can just open up the **package** we want with the `library()` function! In the example of `dplyr`, we do the below.

```
library("dplyr")
```

```
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Now, we can use all the functions in the `dplyr` package! And access any datasets, too! What's in `dplyr` you ask?

```
ls("package:dplyr")[1:50]
```

```
##  [1] "%>%"               "across"            "add_count"
##  [4] "add_count_"        "add_row"           "add_rownames"
##  [7] "add_tally"         "add_tally_"        "all_equal"
## [10] "all_of"            "all_vars"          "anti_join"
## [13] "any_of"            "any_vars"          "arrange"
## [16] "arrange_"          "arrange_all"       "arrange_at"
## [19] "arrange_if"        "as.tbl"            "as_data_frame"
## [22] "as_label"          "as_tibble"         "auto_copy"
## [25] "band_instruments"  "band_instruments2" "band_members"
## [28] "bench_tbls"        "between"           "bind_cols"
## [31] "bind_rows"         "c_across"          "case_match"
## [34] "case_when"         "changes"           "check_dbplyr"
## [37] "coalesce"          "collapse"          "collect"
## [40] "combine"           "common_by"         "compare_tbls"
## [43] "compare_tbls2"     "compute"           "consecutive_id"
## [46] "contains"          "copy_to"           "count"
## [49] "count_"            "cross_join"
```

There is *a lot* in `dplyr`. In this case I truncated it to *only* the first 50 because any more would be a hassle to print. If you ever want to look at what's in any package, you can run similar code (replacing `dplyr` with the other package's name) and get a full list! Impress your friends and confound your enemies with the sheer volume!

In that huge list, there may be only one **object** or **function** you want to use, though, and bringing in *all* of those other ones really clogs up tab-completion. If you want to just use *one* **function** or dataset from a specific library, you can use a double colon `::`. There's an example below with the `MASS` package, looking at various animals' body and brain masses. Neat!

```
MASS::Animals
```

```
##                        body  brain
## Mountain beaver       1.350    8.1
## Cow                 465.000  423.0
```

```
## Grey wolf           36.330  119.5
## Goat                27.660  115.0
## Guinea pig           1.040    5.5
## Dipliodocus      11700.000   50.0
## Asian elephant    2547.000 4603.0
## Donkey             187.100  419.0
## Horse              521.000  655.0
## Potar monkey        10.000  115.0
## Cat                  3.300   25.6
## Giraffe            529.000  680.0
## Gorilla            207.000  406.0
## Human               62.000 1320.0
## African elephant  6654.000 5712.0
## Triceratops       9400.000   70.0
## Rhesus monkey        6.800  179.0
## Kangaroo            35.000   56.0
## Golden hamster       0.120    1.0
## Mouse                0.023    0.4
## Rabbit               2.500   12.1
## Sheep               55.500  175.0
## Jaguar             100.000  157.0
## Chimpanzee          52.160  440.0
## Rat                  0.280    1.9
## Brachiosaurus    87000.000  154.5
## Mole                 0.122    3.0
## Pig                192.000  180.0
```

`MASS` also has a lot of really useful functions in it and if you continue in statistics, sooner or later you will make more full use of it!