How does R handle overlapping object names?

Authored by

stats writer

June 30, 2024

R is a programming language that allows for the creation and manipulation of objects. In some cases, it is possible for two or more objects to have the same name, which can lead to confusion and errors. To handle this issue, R uses a scoping system that determines which object should be used based on its location or environment. This means that when there are overlapping object names, R will prioritize the object that is located in the current environment or is closest in the search path. This allows for multiple objects with the same name to exist within the same program without causing conflicts. Additionally, R also has the ability to assign unique names to objects that have the same name, ensuring that all objects can be accessed and manipulated accurately. Overall, R's handling of overlapping object names ensures efficient and organized programming.

How does R handle overlapping object names? | R FAQ

Because R is developed by an open source community, it is not uncommon
that multiple packages may use the same name for a function or dataset.
If you load packages that use the same name for an object, R will warn that
certain object(s) have been "masked".

Spotting the Problem

Masking occurs when two or more packages have objects (such as functions) with
the same name. Here is an example.

require(plyr)require(Hmisc)

Dealing with Masking

Most of the time, masking is not a problem. However, there are times when
you need to load multiple packages and a function you want to use becomes masked
or you may even need to use a function with the same name from both packages.
In these cases, there are several possibilities.

If you know you want to use the summarize function in plyr, you can
use a double colon to qualify the reference. For example:

```
plyr::summarize

function (.data, ...)
{

#output omitted


}
<environment: namespace:plyr>
```

Here you can see that the function definition is returned, and there is a note
indicating that the environment is the namespace of the plyr package.
The general form is package::function, so if we wanted to use the
summarize function from the Hmisc package:

```
Hmisc::summarize

function (X, by, FUN, ..., stat.name = deparse(substitute(X)),
type = c("variables", "matrix"), subset = TRUE, keepcolnames = FALSE)
{
#output omitted
}
<environment: namespace:Hmisc>
## use summarize function from plyr() rather than Hmisc()
## to calculate average weight of a car

plyr::summarize(mtcars,avgwt = mean(wt))

avgwt
```

1 3.21725

Note that if we just type summarize, R will return the value from
the Hmisc package, because that package was loaded after
the plyr package.

Typically, judicious loading of packages is preferred over explicit references.
For day-to-day use, it is also common to load the package with the function you want
to use last so that it's definition of the function masks those of the package(s)
you do not want to use.

Another possibility is to use the get function. The get function takes
as the first argument the name of the object (a function in this case) and then the
location or environment of the object. Going back to our rename example:

```
get(x = "summarize", pos = "package:plyr")
```

```
function (.data, ...)
{
#output omitted
}
<environment: namespace:plyr>
```

It is more cumbersome, but a perfectly clear way to clarify where to get something. See the documentation for ?get for further details.

Technical Notes

When a package is loaded via library or require, it is automatically attached to the search path (typically in position 2). The first position is reserved for the global environment (i.e., the workspace where interactive commands are executed). When an object (function, data, etc.) is called from the command line, R first looks in the global environment, then proceeds up the search path until the object is found. If no instances exist, R reports that the object could not be found.

The search path can be viewed using the search function. For example:

search()

".GlobalEnv" "package:Hmisc" "package:ggplot2" "package:Formula"
"package:survival" "package:lattice" "package:plyr" "tools:rstudio"
"package:stats" "package:graphics" "package:grDevices" "package:utils"
"package:datasets" "package:methods" "Autoloads" "package:base"

The order indicates the objects are searched for. Two similar functions, .packages and loadedNamespaces return a character vector of the names of loaded packages or namespaces, respectively. For example, in the session used to create this page:

(.packages())

"Hmisc" "ggplot2" "Formula" "survival" "lattice" "plyr" "stats"
"graphics" "grDevices" "utils" "datasets" "methods" "base"

loadedNamespaces()

"Rcpp" "lattice" "grDevices" "Hmisc" "chron" "grid"
"plyr" "gtable" "acepack" "datasets" "scales" "ggplot2"
"utils" "data.table" "latticeExtra" "rpart" "Matrix"
"graphics"
"base" "Formula" "splines" "RColorBrewer" "tools"
"foreign"
"munsell" "survival" "stats" "colorspace" "cluster"
"nnet"
"gridExtra" "methods"

Issues with masking and unclear references are one (of several) reason why it is not advised
to use attach on a dataset. When a dataset is attached, it is added to the search
path. As the number of variables in the dataset increases, the chance that some function
or object in a package will be masked. Further, suppose that after attaching a dataset,
a package is loaded. By default, the package gets added in position 2, while the dataset becomes
position 3. Now if you call detach, it is the package, not the dataset that will be

unattached. This is demonstrated below.

attach(mtcars)search()

**".GlobalEnv" "mtcars" "package:Hmisc" "package:ggplot2"
"package:Formula" "package:survival" "package:lattice" "package:plyr"
"tools:rstudio" "package:stats" "package:graphics" "package:grDevices"
"package:utils" "package:datasets" "package:methods" "Autoloads"
"package:base"**

require(MASS)search()

**".GlobalEnv" "package:MASS""mtcars" "package:Hmisc"
"package:ggplot2" "package:Formula" "package:survival" "package:lattice"
"package:plyr" "tools:rstudio" "package:stats" "package:graphics"
"package:grDevices" "package:utils" "package:datasets" "package:methods"**

"Autoloads" "package:base"

detach()search()

".GlobalEnv" "mtcars" "package:Hmisc" "package:ggplot2"
"package:Formula" "package:survival" "package:lattice" "package:plyr"
"tools:rstudio" "package:stats" "package:graphics" "package:grDevices"
"package:utils" "package:datasets" "package:methods" "Autoloads"
"package:base"

Special Problems and Advanced References

Because of special environments for packages, masking is usually only a problem for users, not for packages themselves. However, if you notice a package is having problems finding the correct function, you should contact the package maintainer. Properly written packages do not have problems regardless of what other packages you attach, and the fix is straight forward for the package maintainer. To find out who the current

maintainer is, you can use:

maintainer("plyr")

"Hadley Wickham <hadley@rstudio.com>"

where the first argument to the function is the name of the package in quotes.
Note that the maintainer may be different from the author, and the maintainer is the correct individual to contact.

Partly to reduce confusion and problems of masking, R packages have a namespace. This is just a special environment (you can think of it like a filing cabinet) that belongs to that package. Authors have a choice what functions or objects from their package are made publicly available and which are only available internally. With masked objects, the problem was that the wrong object was used. With internal (non exported) objects,
R will say that it cannot find the object. If you want to look at or use an internal object, you can use the triple colon operator, ":::". Note that ":::" also qualifies the

reference with the package name (as the double colons "::" did), so there can be no confusion about where it came from. Below is an example using the plyr package. id_var is an internal function in plyr, to access it, we would need to use the package:::function syntax, as below.

```
id_var

## Error: object 'id_var' not found
plyr:::id_var

function (x, drop = FALSE)
{
#output omitted
}
<environment: namespace:plyr>
```