Introduction to Defaults

Jeffrey A. Ryan
jeff.a.ryan@gmail.com

July 25, 2007

1 Introduction

A common problem when writing functions for use by others is deciding upon sensible default values that will appeal to most of your target user's daily needs.

As an end user, often these defaults may not fit the problem at hand, and will require some fine-tuning to make the function perform as desired. Additionally, while all values may be changed within an actual function call, it may not always be desirable to have to remember the new defaults, or to re-enter them with each function call.

The Defaults [1] package allows the end user to pre-specify a default for any formal argument in a given function, and to *force* the function to use these defaults. For the function authors it is no longer necessary to hand-code checks to R's internal options, as the addition of one function call at the beginning of any function needing access to user specified defaults will manage the process for them.

This document will cover the Defaults package from the perspective of the user and the developer. We begin with how the R end user can benefit from using Defaults.

2 The End User

Every person using R, whether for analysis or as a developer, is an end user. Countless functions are used within a typical session, often with multiple optional argument settings for each. One of the most common is `ls`. If one would like to have `ls` display even hidden objects in a given environment it is necessary to add the argument `all.names=TRUE` to the call. With Defaults, one can specify outside the function call this new default value, so that subsequent calls will now display all names.

```
> hello <- "visible"
> .goodbye <- "hidden"
> ls()

[1] "hello"

> library(Defaults)
> setDefaults("ls", all.names = TRUE)
> useDefaults(ls)
> ls()

[1] ".goodbye" "hello"    "ls"

> ls(all.names = FALSE)

[1] "hello" "ls"
```

After loading the Defaults library, a call to `setDefaults('ls',all.names=TRUE)` is made. This creates an entry in the standard options list, with the name `ls.Default`, attaching the value all.names=TRUE to this entry.

At this point the original function `ls` is unable to process this new user specified default. By calling `useDefaults(ls)` a new copy of `ls` is added to the user's workspace, with the notable difference that this copy *can* process the new defaults. The original function is effectively hidden from the user, allowing the Defaults functionality to be used. In the event that the function is *already* in the user's workspace (e.g. a user defined function) a copy of the original is made and hidden from normal view.

Internally, the function first looks to see if any arguments have been specified in the actual function call, as these take precedence over *any* default - formal or via `setDefaults`. If no value is given in the call, the global defaults, if any, are checked. If nothing is still set, the process falls back to the original formal defaults, if any, and continues executing.

At present it is *NOT* possible to set an argument's value as `NULL` or to a function. Values that cannot be set via `setDefaults` may of course still be specified in the function call.

The set defaults can be viewed with `getDefaults`, and unset with a call to `unsetDefaults`. The former, when called with no arguments, will return a character vector of all functions currently having defaults set for use with Defaults.

```
> getDefaults(ls)
```

```
$name
NULL

$pos
NULL

$envir
NULL

$all.names
[1] TRUE

$pattern
NULL
```

```
> getDefaults()
```

```
[1] "ls"
```

```
> unsetDefaults(ls, confirm = FALSE)
```

Since using global default as an end user is so easy, it only makes sense that making use of them as a developer would be just as straighforward. It's even easier.

3 The Developer

Without the Defaults package, if one is to use a mechanism to access globally specified defaults, designed specifically for a new function, it would require a complete lookup facility, as well as a series of `if-else` blocks. With Defaults all that is required is one function placed at the beginning of your function.

```
> fun <- function(x = 5, y = 5) {
+     importDefaults()
+     x^y
+ }
> fun()
```

```
[1] 3125
```

```
> fun(x = 10)
```

```
[1] 1e+05
```

`importDefaults()` places all *non*-NULL default arguments specified by an earlier call to `setDefaults` into the current function's environment. The only exception would be if the argument has been specified in the function call itself, at which point the value or values in question would *NOT* be loaded into the current scope.

```
> setDefaults(fun, x = 8, y = 2)
> fun()

[1] 64

> fun(9)

[1] 81

> fun(y = 0.5)

[1] 2.828427

> unsetDefaults(fun, confirm = FALSE)
> fun()

[1] 3125
```

4 Conclusion

Using Defaults, whether as an end user or package developer, greatly simplifies the process of utilizing externally set global defaults. With a small set of functions, users can create and use default arguments in place of formal ones, as well as create defaults where none normally exist, all without relying on the underlying function's own methods for handling defaults. Future development may include the ability to use `NULL` as a legal default for the rare occasion that it is desired, as well as a better method of handling subsequent function calls within the visible parent function, as is the case with S3-style method dispatch.

References

[1] Jeffrey A. Ryan: *Defaults: Create Global Function Defaults*, R package version 1.0-5, 2007