Introduction to Defaults

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

August 18, 2007

Contents

1 Introduction

A common problem when writing functions for use by others is deciding upon sensible argument defaults 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 [4] package allows the end user to pre-specify a default value 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 perspectives of the user and the developer, as well as give some detail as to its implementation. We begin with how the R end user can benefit from using Defaults.

2 The End User: `setDefaults`

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` *always* display even hidden objects in a given environment it is necessary to add the argument `all.names=TRUE` to the call. By using the Defaults package, 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)
> ls()

[1] ".goodbye" "hello"

> ls(all.names = FALSE)

[1] "hello"

> unDefaults(ls)
> ls()

[1] "hello"
```

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. `setDefaults` then calls `useDefaults(ls)` internally, and a change is then made to `ls` internally, with the important difference being that now this new version *can* process the new defaults. The original function is modified in its original environment to allow the Defaults functionality to be used.

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 set 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. A more detailed account of the exact changes made and how everything works is describe later in this document.

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

To remove a specific default argument, simply set it to NULL within a call to `setDefaults`.

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, though these will not *necessarily* be currently set to use Defaults (i.e. a call would still need to be made to `useDefaults` to enable Defaults functionality in situations where a user call to `unDefaults` had removed it). `unsetDefaults` also removes Defaults functionality, by calling `unDefaults` internally.

```
> getDefaults(ls)

$all.names
[1] TRUE

> getDefaults()

[1] "ls"

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

Since using Defaults 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: using `importDefaults`

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. Or for the truly lazy developer, no change at all – as the end user can always add this functionality (see **The End User**).

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

```
[1] 25
```

```
> fun(x = 1:5)
```

```
[1]  5 10 15 20 25
```

`importDefaults("fun")` 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 had been specified in the function call itself, at which point the value or values in question would *NOT* be loaded into the current scope. In other words, if you explicitly specify an argument value in the function call, *that* is the value that will be used by the function.

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

```
[1] 16
```

```
> fun(9)
```

```
[1] 18
```

```
> fun(y = 0.5)
```

```
[1] 4
```

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

```
[1] 25
```

4 How it works internally

A little background on just how this all works, for those who care to know the details.

4.1 Where defaults are stored

All options set via `setDefaults` are store in the standard R `options` list, using a special naming convention of appending `.Default` to the function in question. This also means that values are lost from one session to another, which is designed to prevent unintentional overrides in subsequent sessions.

4.2 How functions gain access

Calling `setDefaults` (which calls `useDefaults` internally) on a function magically changes the function to handle your previously specified defaults. Actually, there isn't much magic. `useDefaults` modifies the function in question (actually the first instance of said function in its search path), by inserting at the start of the function a call to `importDefaults`. The function is modified in-place to allow for namespaces to resolve correctly, as well as to prevent workspace clutter and assure reversion to its original state in subsequent R sessions. The gory details of this process can be viewed in the source, and have been greatly influenced by comments from John Chambers [2] and the source code of `trace` [3] and Mark Bravington's `mtrace` from his debug [1] package.

5 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 on non-visible methods.

References

[1] Mark V. Bravington: *debug: MVB's debugger for R*, R package version 1.1.0, 2005

[2] John M. Chambers: *Personal Communication*, useR! 2007, 9 August, 2007

[3] R Development Core Team: *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org

[4] Jeffrey A. Ryan: *Defaults: Create Global Function Defaults*, R package version 1.1-0, 2007