

# Package ‘Defaults’

April 17, 2009

**Type** Package  
**Title** Create Global Function Defaults  
**Version** 1.1-1  
**Date** 2007-08-17  
**Author** Jeffrey A. Ryan  
**Maintainer** Jeffrey A. Ryan <jeff.a.ryan@gmail.com>  
**Description** Set, Get, and Import Global Function Defaults  
**LazyLoad** yes  
**License** GPL (>= 2)  
**Repository** CRAN  
**Date/Publication** 2007-08-26 15:20:34

## R topics documented:

|                            |   |
|----------------------------|---|
| Defaults-package . . . . . | 1 |
| getDefaults . . . . .      | 3 |
| importDefaults . . . . .   | 4 |
| setDefault . . . . .       | 6 |
| useDefaults . . . . .      | 8 |

|              |           |
|--------------|-----------|
| <b>Index</b> | <b>11</b> |
|--------------|-----------|

---

Defaults-package    *Create Global Function Defaults*

---

## Description

Set, Get, and Import Global Function Defaults

## Details

```

Package: Defaults
Type: Package
Version: 1.1-1
Date: 2007-08-17
LazyLoad: yes
License: GPL 2 or later
Packaged: Sun Jul 25 10:03:19 2007; jryan
Built: R 2.5.0; ; 2007-07-25 10:03:38; unix

```

### Index:

```

useDefaults          Enable Global Default Check by Function
unDefaults          Disable Global Default Check by Function

getDefaults          Show Global Defaults List by Function
setDefaults          Create Global Defaults List by Function
unsetDefaults        Remove All Global Defaults by Function

importDefaults        Import Global Default Argument Values

```

**Defaults** makes the use of globally specified defaults, on a per-function basis, available to *all* functions written in R.

Through the use of `importDefaults` which can be hard coded into a function by the function author, or dynamically enabled for *any* visible function through `useDefaults`, a user can now override the author specified defaults as returned by `formals` by a call to `setDefaults`.

### Note

It is important to note that only exported functions may have defaults set, which *should not* be an issue, as they are the only user callable functions.

A big thank you to John Chambers and Dirk Eddelbuettel who pointed out my mishandling of functions with namespaces calling internal non-exported functions. A new implementation of `useDefaults` now handles this correctly.

### Author(s)

Jeffrey A. Ryan  
 Maintainer: Jeff Ryan <jeff.a.ryan@gmail.com>

### Examples

```

formals(sd)          # what _can_ be set
try(sd(c(1:10,NA)), silent=TRUE) # fails

useDefaults(sd)     # not necessary - setDefaults will call
setDefaults(sd, na.rm=TRUE)
getDefaults(sd)     # what _has_ been set

```

```
sd(c(1:10,NA)) # works!

unsetDefaults(sd, confirm=FALSE) # removes previously set default
unDefaults(sd) # unnecessary, as unsetDefaults calls automatically
```

---

getDefaults *Show Global Defaults List By Function*

---

## Description

Show global Default list by function or specified argument values by function.

## Usage

```
getDefaults(name=NULL, arg = NULL)
```

## Arguments

|      |                                      |
|------|--------------------------------------|
| name | name of function, quoted or unquoted |
| arg  | values to retrieve                   |

## Details

A list of function names currently with Defaults set can be seen by calling `getDefaults()` with no arguments.

This *does not* imply that the returned function names are currently set to accept defaults (via `useDefaults` or hard-coded with `importDefaults`, rather that they have been set up to store user Defaults.

All values can be viewed less elegantly by a call to `getOption(name_of_function.Default)`

## Value

`getDefaults` returns a named list of defaults and associated values, similar to `formals`, only returning `setDefault`s set values for the name function. Single arguments need not be quoted, multiples must be as a character vector. Calling `getDefaults()` without arguments results in a character vector of all functions currently having Defaults set (by `setDefault`s)

## Author(s)

Jeffrey A. Ryan

## See Also

[setDefault](#), [useDefaults](#), [options](#)

**Examples**

```

setDefaultts(lm, na.action='na.exclude', singular.ok=TRUE)

getDefaultts()
getDefaultts(lm)

unsetDefaultts(lm, confirm=FALSE)
getDefaultts(lm)

my.fun <- function(x=2, y=1) { x ^ y }
my.fun()           #returns 2
my.fun(x=2, y=10) #returns 1024

setDefaultts(my.fun, x=2, y=3)
useDefaultts(my.fun)
my.fun

my.fun()           #returns 8
my.fun(y=10)       #returns 1024
my.fun(x=2, y=10) #returns 1024

unDefaultts(my.fun)
my.fun
my.fun()           #returns 2

getDefaultts(my.fun)
unsetDefaultts(my.fun, confirm=FALSE)
getDefaultts(my.fun)

```

---

importDefaults      *Import Global Default Argument Values*

---

**Description**

Use globally specified defaults, if set, in place of formally specified default argument values. Allows user to specify function defaults different than formally supplied values, e.g. to change poorly performing defaults, or satisfy a different preference.

**Usage**

```
importDefaults(calling.fun)
```

**Arguments**

calling.fun    name of function to act upon

## Details

Placed immediately after the function declaration, a call to `importDefaults` checks the user's environment for globally specified default values for the called function. These defaults can be specified by the user with a call to `setDefault`s, and will override any default formal parameters, in effect replacing the original defaults with user supplied values instead.

If a function has *not* been written with `importDefaults` (most R functions, so far...), it is possible to simply call `useDefaults` to achieve the same results. As of version 1.1-0, simply calling `setDefault`s will call `useDefaults` internally, removing the need to explicitly call. See the related help page.

Any values specified by the user in a in the parent function (that is, the function containing `importDefaults`) will override the values set in the global default environment.

## Value

Used for its side-effects, a call to `importDefaults` loads all non-NULL default values specified by the user into the current function's environment, effectively changing the default values passed in the parent function call.

`importDefaults` values, like formally defined defaults in the function definition, take lower precedence than arguments specified by the user in the function call.

An alias to `importDefaults`, `.importDefaults` is the actual function added when `useDefaults` is called on a function. The naming convention is designed to facilitate setting and unsetting, and should *NOT* be used by the function developer. Please use only `importDefaults`

## Note

It is important to note that when a function implements `importDefaults`, non-named arguments *may* be ignored if a global Default has been set (i.e. not NULL).

If this is the case, simply name the arguments in the calling function.

This *should* also work for functions retrieving formal parameter values from `options`, as it assigns a value to the parameter in a way that looks like it was passed in the function call. So any check on `options` would presumably disregard `importDefaults` values if an argument was passed to the function (what `useDefaults` does)

## Author(s)

Jeffrey A. Ryan

## See Also

[useDefaults](#), [setDefault](#)s

## Examples

```
my.fun <- function(x=3)
{
  importDefaults('my.fun')
  x ^ 2
}
```

```

}

my.fun()          #returns 9

setDefaults(my.fun, x=10)
my.fun()          #returns 100
my.fun(x=4)       #returns 16

getDefaults(my.fun)
formals(my.fun)
unsetDefaults(my.fun, confirm=FALSE)
getDefaults(my.fun)

my.fun()          #returns 9

```

---

setDefaults                      *Create Global Defaults List By Function*

---

## Description

Create Defaults to be used in place of a function's formal argument values.

In conjunction with `useDefaults`, or functions already supporting `importDefaults`, to allow for user override of formal arguments without specifying in the function call.

See `useDefaults` for the details of this process.

## Usage

```

setDefaults(.name, ...)
unsetDefaults(name, confirm=TRUE)

```

## Arguments

|                      |                                      |
|----------------------|--------------------------------------|
| <code>.name</code>   | name of function, quoted or unquoted |
| <code>name</code>    | name of function, quoted or unquoted |
| <code>...</code>     | name=value default pairs             |
| <code>confirm</code> | prompt before unsetting defaults     |

## Details

`setDefaults` is a wrapper to `R options`, allowing the user to specify any name=value pair that a function has in its formal arguments.

Calling `setDefaults` on an object that previously could not process Defaults will automatically enable defaults via an internal call to `useDefaults`.

Only formal name=value pairs specified will be updated. Values do not have to be respecified in subsequent calls to `setDefaults`, so it is possible to add new defaults one at a time for each

function, without having to retype all previous values. Assigning NULL to any argument will remove the argument from the Defaults list.

When a function is set to use these Defaults (using `useDefaults`, `setDefaults`, or hard-coded to use the **Defaults** package) all non-NULL values set by `setDefaults` will effectively replace all formally specified function defaults.

At present it is not possible to specify NULL as a replacement for a non-NULL default, as the process interprets NULL values as being not set, and will simply use the value specified formally in the function. If NULL is what is desired, it is necessary to include this in the function call itself.

Any arguments included in the actual function call will take precedence over `setDefaults` values, as well as the standard formal function values. This conforms to the current user experience in R.

`unsetDefaults` does exactly what it says.

### Value

None. Called for its side effect of setting a list of Default arguments by function.

### Note

Like `options`, settings are *NOT* kept across sessions.

Currently, it is *NOT* possible to pass values for ... arguments, only formally specified arguments in the original function definition.

If it is desired to pass additional arguments, or more specifically have new defaults, to subsequent methods/calls, a separate `setDefaults` `useDefaults` series is required.

`unsetDefaults` removes the *all* entries from the `options` lists for the specified function, and then calls `unDefaults`. To remove single function Default values simply set the name of the argument to NULL in `setDefaults`

### Author(s)

Jeffrey A. Ryan

### See Also

`options`, `getDefaults`, `useDefaults`,

### Examples

```
my.fun <- function(x=2,y=1) { x ^ y }
my.fun()           #returns 2
my.fun(x=2,y=10)  #returns 1024

setDefaults(my.fun, x=2,y=3)
useDefaults(my.fun)
my.fun

my.fun()           #returns 8
my.fun(y=10)      #returns 1024
```

```

my.fun(x=2,y=10)      #returns 1024

unDefaults(my.fun)
my.fun
my.fun()             #returns 2

getDefaults(my.fun)
setDefaults(my.fun,x=NULL) #removes the value for x, leaving just y
unsetDefaults(my.fun,confirm=FALSE)
getDefaults(my.fun)

```

---

useDefaults

*Enable and Disable Global Defaults By Function*


---

## Description

Allows for the use of globally managed default values for formal function arguments. Adds the ability to pre-specify a value for any formal argument as if it were specified in the function call.

## Usage

```

useDefaults(name)
unDefaults(name)

```

## Arguments

name                    name of function, quoted or unquoted

## Details

These functions are called automatically during calls to `setDefaults` and `unsetDefaults`, though they may be called by the user as well.

Defaults are set inside the named function with a call to `importDefaults`. This may be hard coded into the function by the author, or may be dynamically added with a call to `useDefaults`.

Internally, a new call to `importDefaults` is added before the body of the function name. This is added in the first occurrence of the specified function encountered in the search path. That is, if there are two function objects, the first encountered will be modified. The modification takes place in the environment of the original function, so namespaces are retained.

`useDefaults` replaces all formal functional arguments with all non-NULL globally specified ones after first checking that these global defaults have not been overridden with new values in the function call.

The order of lookup is as follows, with the lookup halted once a specified value is found:

1. Check for arguments specified in the actual call
2. Check for arguments specified by `setDefaults`
3. Use original function defaults. (if any)

Setting default values is accomplished via `setDefault`s, with the values being written to R's `options` list as a named list set to the function's name appended with a `.Default`, all managed automatically. It is possible to view and delete all defaults with the functions `getDefault`s and `unsetDefault`s, respectively. All R objects can be saved to the Defaults list, with the exception of `NULL`, as this removes the argument from the Defaults list instead.

To return a function enabled by `useDefaults` to its original state, call `unDefault`s. Conceptually this is similar to `debug` and `undebug`, though implemented entirely in R. The current implementation borrows from the R function `trace` and more directly, Mark V. Bravington's `mtrace`.

### Value

None. Called for its side effect of enabling or disabling the Defaults mechanism. The only use visible side-effect is the modified function body.

### Note

The underlying `importDefault`s mechanism relies on the calling function to have the same name as function in which it is located. This is the case in almost all circumstances, excepting one - when called as the passed FUN object in an `lapply` or similar call, as the calling function will then simply be 'FUN' or something similar. In these circumstances the function will behave as if `useDefaults` had *not* been called on it, i.e. no check of global Defaults will occur. If Defaults behavior is desired, simply create an anonymous function wrapper to the function in question, as this will then resolve correctly.

A special thanks to John Chambers and Dirk Eddelbuettel for providing guidance on handling functions using namespaces, as well as pointing out the original mishandling of namespace issues.

### Author(s)

Jeffrey A. Ryan

### References

Mark V. Bravington (2005) *debug: MVB's debugger for R*, R package version 1.1.0

### See Also

[importDefault](#)s, [setDefault](#)s, [formals](#), [body](#), [as.function](#)

### Examples

```
my.fun <- function(x=2,y=1) { x ^ y }
my.fun()           #returns 2
my.fun(x=2,y=10)  #returns 1024

setDefault(my.fun, x=2, y=3)
useDefault(my.fun)
my.fun

my.fun()           #returns 8
my.fun(y=10)      #returns 1024
```

```
my.fun(x=2,y=10)    #returns 1024

unDefaults(my.fun)
my.fun
my.fun()           #returns 2

getDefaults(my.fun)
unsetDefaults(my.fun,confirm=FALSE)
getDefaults(my.fun)
```

# Index

## \*Topic **package**

Defaults-package, 1

## \*Topic **utilities**

getDefaults, 3

importDefaults, 4

setDefault, 6

useDefaults, 8

.importDefaults (*importDefaults*),  
4

as.function, 9

body, 9

Defaults (*Defaults-package*), 1

Defaults-package, 1

formals, 9

getDefaults, 3, 7

importDefaults, 4, 9

options, 3, 7

setDefault, 3, 5, 6, 9

unDefaults (*useDefaults*), 8

unsetDefaults (*setDefault*), 6

useDefaults, 3, 5, 7, 8