Package 'mvbutils'

October 13, 2022

Version 2.8.232

Author Mark V. Bravington <mark.bravington@csiro.au>

Maintainer Mark V. Bravington <mark.bravington@csiro.au>

Depends R (>= 3.3)

Imports utils, tools, stats, graphics

Date 2018-12-11

Title Workspace Organization, Code and Documentation Editing, Package Prep and Editing, Etc

NeedsCompilation no

ByteCompile no

Description Hierarchical workspace tree, code editing and backup, easy package prep, editing of packages while loaded, per-object lazy-loading, easy documentation, macro functions, and miscellaneous utilities. Needed by debug package.

License GPL (>= 2)

Repository CRAN

Date/Publication 2018-12-12 15:30:03 UTC

R topics documented:

mvbutils-package 3
cd 6
cdfind 10
cdprompt 12
changed.funs 13
check.patch.versions 13
ditto.list 14
do.in.envir 15
do.on 16
doc2Rd 17
dochelp 23
dont.lock.me 24
dont.lockBindings 25

extract.named . . . 26
fast.read.fwf . . . 27
find.documented . . . 27
fix.order . . . 29
fixr . . . 30
flatdoc . . . 36
foodweb . . . 38
generic.dll.loader . . . 41
get.backup . . . 43
hack . . . 45
help . . . 46
help2flatdoc . . . 49
install.pkg . . . 50
library.dynam.reg . . . 53
local.on.exit . . . 54
local.return . . . 55
lsize . . . 56
maintain.packages . . . 57
make.NAMESPACE . . . 59
make_dull . . . 60
max_pkg_ver . . . 61
mcut . . . 62
mlazy . . . 63
mlocal . . . 67
move . . . 69
multirep . . . 71
mvb.sys.parent . . . 72
mvbutils.operators . . . 74
mvbutils.packaging.tools . . . 76
mvbutils.utils . . . 83
my.index . . . 89
NEG . . . 90
noice . . . 91
pre.install . . . 92
print . . . 100
rbdf . . . 104
readLines.mvb . . . 108
rm.pkg . . . 109
Save . . . 110
search.for.regexpr . . . 112
set.finalizer . . . 113
setup.mcache . . . 114
sleuth . . . 115
source.mvb . . . 116
strip.missing . . . 118
task.home . . . 119
unpackage . . . 120
warn.and.subset . . . 121

Index **123**

`mvbutils-package` *How to use the mvbutils package*

Description

Package **mvbutils** is a collection of utilities offering the following main features:

- Hierarchical organization of projects (AKA tasks) and sub-tasks, allowing switching within a single R session, searching and moving objects through the hierarchy, objects in ancestor tasks always visible from child (sub)tasks, etc. See `cd`.
- Improved function, text, and script editing facilities, interfacing with whichever text editor you prefer. The R command line is not frozen while editing, and you can have multiple edit windows open. Scriptlets can be edited as expressions, for subsequent calls to `eval`. Function documentation can be stored as plain text after the function definition, and will be found by `help` even if the function isn't part of a package. There is also a complete automatic text-format backup system for functions & text. See `fixr`.
- Automated package construction, including production of Rd-format from plain text documentation. Packages can be edited & updated while loaded, without needing to quit/rebuild/reinstall. See `mvbutils.packaging.tools`.
- "Lazy loading" for individual objects, allowing fast and transparent access to collections of biggish objects where only a few objects are used at a time. See `mlazy`.
- Miscellaneous goodies: local/nested functions (`mlocal`), display of what-calls-what (`foodweb`), multiple replacement (`multirep`), nicely-formatted latex tables (`xtable.mvb`), numerous lower-level lower-level utility functions and operators (`mvbutils.utils`, `mvbutils.operators`, `extract.named`, `mcut`, `search.for.regexpr`, `strip.missing`, `FOR`)

To get the full features of the **mvbutils** package– in particular, the project organization– you need to start R in the same directory every time (your "ROOT task"), and then switch to whichever project from inside R; see `cd`. Various `options` always need to be set to make `fixr` and the **debug** package work the way you want, so one advantage of the start-in-the-same directory-approach is that you can keep all your project-independent `options()`, library loads, etc., in a single `.First` function or ".Rprofile" file, to be called automatically when you start R. However, many features (including support for the **debug** package) will work even if you don't follow this suggestion.

The remaining sections of this document cover details that most users don't know about; there's no need to read them when you are just starting out with `mvbutils`.

Housekeeping info

On loading, the **mvbutils** package creates a new environment in the search path, called `mvb.session.info`, which stores some housekeeping information. `mvb.session.info` is never written to disk, and disappears when the R session finishes. [For Splus users: `mvb.session.info` is similar to frame 0.] You should never change anything in `mvb.session.info` by hand, but it is sometimes useful to look at some of the variables there:

- `.First.top.search` is the directory R started in (your ROOT task).

- .Path shows the currently-attached part of the task hierarchy.
- base.xxx is the original copy of an overwritten system function, e.g. library
- fix.list keeps track of objects being edited via fixr
- session.start.time is the value of Sys.time() when mvbutils was loaded
- source.list is used by source.mvb to allow nesting of sources
- r.window.handle is used by the **handy** package (Windows only)
- partial.namespaces is used to alleviate difficulties with unloadable data files– see mvbutils.packaging.tools
- things whose name starts with ".." are environments used in live-editing packages
- maintained.packages is a list of the latter

Redefined functions

On loading, package **mvbutils** redefines a few system functions: lockEnvironment, importIntoEnv loadNamespace, print.function, help, rbind.data.frame and, by default, library, savehistory, loadhistory, and save.image. (The original version of routine xxx can always be obtained via base.xxx if you really need it.) The modifications, which are undone when you unload mvbutils, should have [almost] no side-effects. Briefly:

- library is modified so that its default pos argument is just under the ROOT workspace (the one that was on top when mvbutils was loaded), which is needed by cd. This means that packages no longer get attached by default always in position 2.
- lockEnvironment and importIntoEnv are modified to allow live-editing of your own maintained packages– no change to default behaviour.
- loadNamespace has the default value of its "partial" argument altered, to let you bypass .onLoad for selected faulty packages– see mvbutils.packaging.tools and look for partial.namespaces. This allows the loading of certain ".RData" files which otherwise crash from hidden attempts to load a namespace. It lets you get round some truly horrendous problems arising from faults with 3rd-party packages, as well as problems when you stuff up your own packages.
- rbind.data.frame does not ignore zero-row arguments (so it takes account of their factor levels, for example).
- rbind.data.frame: dimensioned elements (i.e. matrices & arrays within data.frames) no longer have any extra attributes removed. Hence, for example, you can (if you are also using my nicetime package) rbind two data frames that both have POSIXct-matrix elements without turning them into raw seconds and losing timezones.
- help and ? are modified so that, if utils:::help can't find help for a function (but not a method, dataset, or package), it will look instead for a doc attribute of the function to display in a pager/browser using dochelp. Character objects with a ".doc" extension will also be found and displayed. This lets you write and distribute "informal help".
- loadhistory and savehistory are modified so that they use the *current* "R_HISTFILE" environment variable if it set. This can be set dynamically during an R session using Sys.setenv. Standard R behaviour is to respect "R_HISTFILE" iff it is set *before* the R session starts, but not to track it during a session. If "R_HISTFILE" is not set, then cd will on first use set "R_HISTFILE" to "«ROOT task»/.RHistory", so that same the history file will be used throughout each and every session.

- save.image is modified to call Save instead; this will behave exactly the same for workspaces not using mvbutils task-hierarchy feature or the **debug** package, but otherwise will prevent problems with mtraced functions and mlazyed objects.
- print.function is modified to let you go on seamlessly using functions written prior to R 2.14 in conjunction with the srcref system imposed by R 2.14; see fixr.

Some of these redefinitions are optional and can be turned off if you really want: loadhistory, savehistory, save.image, library, lockEnvironment, importIntoEnv, and loadNamespace. To turn them off, set options(mvbutils.replacements=FALSE) *before* loading mvbutils. However, I really don't recommend doing so; it will prevent cd etc, fixr, and the package-maintenance tools from working properly, and if you use debug you will probably cause yourself trouble when you forgetfully save.image an mtraced function. You can also set the "mvbutils.replacements" option to a character vector comprising some or all of the above names, so that only those happen; if so, you're on your own. The other replacements are unavoidable (but should not be apparent for packages that don't import mvbutils).

After mvbutils has loaded, you can undo the modification of a function xxx by calling assign.to.base("xxx", base.xxx). Exceptions are help, ?, print.function, rbind.data.frame which are intrinsic to mvbutils. Unloading mvbutils' will undo all the modifications.

Nicer posixt behaviour: POSIXct etc have some nasty behaviour, and mvbutils used to include some functions that ameliorated things. I've moved them into a separate package **nicetime**, available on request.

Ess and mvbutils

For ESS users: I'm not an Emacs user and so haven't tried ESS with the **mvbutils** package myself, but a read-through of the ESS documentation (as of ~2005) suggests that a couple of ESS variables may need changing to get the two working optimally. Please check the ESS documentation for further details on these points. I will update this helpfile when/if I receive more feedback on what works (though there hasn't been ESS feedback in ~8 years...).

- cd changes the search list, so you may need to alter "ess-change-sp-regex" in ESS.
- cd also changes the prompt, so you may need to alter "inferior-ess-prompt". Prompts have the form WORD1/WORD2/.../WORDn> where WORDx is a letter followed by zero or more letters, underscores, periods, or digits.
- move can add/remove objects in workspaces other than the top one, so if ESS relies on stored internal summaries of "what's where", these may need updating.

Display bugs

If you have a buggy Linux display where readline() always returns the cursor to the start of the line, overwriting any prompt, then try options(cd.extra.CR=TRUE).

Author(s)

Mark Bravington

See Also

cd, fixr, mlazy, flatdoc, dochelp, maintain.packages, source.mvb, mlocal, do.in.envir, foodweb, mvbutils.operators, mvbutils.utils, mvbutils.packaging.tools, package **debug**

---

cd *Organizing R workspaces*

Description

cd allows you to set up and move through a hierarchically-organized set of R workspaces, each corresponding to a directory. While working at any level of the hierarchy, all higher levels are attached on the search path, so you can see objects in the "parents". You can easily switch between workspaces in the same session, you can move objects around in the hierarchy, and you can do several hierarchy-wide things such as searching, even on parts of the hierarchy that aren't currently attached.

Usage

```
# Occasionally: cd()
# Usually: cd(to)
# Rarely:
 cd(to, execute.First = TRUE, execute.Last = TRUE)
```

Arguments

to	the path of a task to move to or create, as an unquoted string. If omitted, you'll be given a menu. See **Details**.
execute.First	should the .First.task code be executed on attachment? Yes, unless there's a bug in it.
execute.Last	should the .Last.task code be executed on detachment? Yes, unless there's a bug in it.

Details

R workspaces can become very cluttered, so that it becomes difficult to keep track of what's what (I have seen workspaces with over 1000 objects in them). If you work on several different projects, it can be awkward to work out where to put "shared" functions– or to remember where things are, if you come back to a project after some months away. And if you just want to test out a bit of code without leaving permanent clutter, but while still being able to "see" your important objects, how do you do it? cd helps with all such problems, by letting you organize all your projects into a single tree structure, regardless of where they are stored on disk. Each workspace is referred to (for historical reasons) as a "task".

Note that there is a basic choice when working with R: do you keep everything you write in a text file which you source every time you start; or do you store all the objects in a workspace as a binary image in a ".RData" file, and rely on save and load? [Hybrids are possible, too.] Some people prefer the text-based approach, but others including me prefer the binary image approach;

my reasons are that binary images let me organize my work across tasks more systematically, and that repeated text-sourcing is much too slow when lengthy analyses or data extractions are involved. The `cd` system is really geared to the binary image model and, before `cd` moves to a new task, either up or down the hierarchy, the current workspace is automatically saved to a binary image. Nevertheless, I don't think `cd` is incompatible with other ways of working, as long as the ".RData" file (actually the `tasks` object) is not destroyed from session to session. At any rate, some people who work by `source`ing large code files still seem to find `cd` useful; it's even possible to use the `.First.task` feature to auto-load a task's source files into a text editor when you `cd` to that task. With the ".RData"-only approach, it is highly advisable to have some way of keeping separate text backups, at least of function code. The `fixr` editing system is geared up to this, and I presume other systems such as ESS are too.

To use the `cd` system, you will need to start R in the **same** workspace every time. This will become your ROOT or home task, from which all other tasks stem. There need not be much in this workspace except for an object called `tasks` (see below), though you can use it for shared functions that you don't want to organize into a package. From the ROOT task, your first action in a new R session will normally be to use `cd` to switch to a real task. The `cd` command is used both to switch between existing tasks, and to create new ones.

To set yourself up for working with `cd`, it's probably a good idea to make the ROOT task a completely new blank workspace, so the first step is to (outside R) create an empty folder with some name like "Rstart". [In MS-Windows, you should think about **where** to put this, to save yourself inordinate typing later on. If you are planning to create a completely new set of folders for your R projects, you might want to put this ROOT folder near the top of the disk directory structure, rather than in the insane default that Windows proffers, which usually looks something like "c:\document...\local...\long...\ridiculous...". However, if you are planning instead to link existing folders into the task hierarchy, then it's better to create the ROOT folder just above, or parallel to, the location of these folders.] Start R in this folder, type `library( mvbutils)`, and then start linking your existing projects into the task hierarchy. [Of course, this assumes that you do have existing projects. If you don't, then just start creating new tasks.] To link in a project, just type `cd()` and a menu will appear. The first time, there will be only one option: "CREATE NEW TASK". Select it (or type 0 to quit if you are feeling nervous), and you will be prompted for a "task name", by which R will always subsequently refer to the task. Keep the name short; it doesn't have to be related to the location of the disk directory where the .RData lives. Avoid spaces and weird characters– use periods as separators. Task names are case-sensitive. Next, you'll be asked which disk directory this task refers to. By default, `cd` expects that you are creating a new task, and therefore suggests putting the directory immediately below the current task directory. However, if you are linking in an existing project, you'll need to supply the directory name. You can save huge amounts of typing by using "." to refer to the current directory, and on *nix systems you can use "~" too. Next, you'll be returned to the R command prompt– but the prompt will have changed, so that the ">" is preceded by the task name. If you type `search()`, you'll see your ROOT task in position 2, below .GlobalEnv as usual. Despite the name, though, the new .GlobalEnv contains the project you've just linked, and if you type `ls()`, you should see some familiar objects. Now type `cd(0)` to move back to the ROOT task (note the changed prompt), type `search()` and `ls()` again to orient yourself, and proceed as before to link the rest of your pre-existing tasks into the hierarchy. When you now type `cd()`, the menu will have more choices. If you select an existing task rather than creating a new one, you will switch straightaway to that workspace; watch the prompt.

Once you have a hierarchy set up, you can switch the current workspace within the hierarchy by calling e.g. `cd(existing.task)` (note the lack of quotes), or by calling `cd()` and picking off the menu. You can move through several levels of the hierarchy at once, using a path specifier such

as `cd(mytask/data/funcs)` or `cd(../child.of.sibling)`. Path specifiers are just like Unix or Windows disk paths with "/" as the separator, so that "." means "current task" and ".." means "parent". However, the character 0 must be used to denote the ROOT task, so that you have to type `cd(0/different.task)` rather than `cd(/different.task)`. You can display the entire hierarchy by calling `cdtree(0)`, or graphically via `plot( cdtree( 0))`.

When you first set up your task hierarchy, you'll also want to create or modify the `.First` function in your ROOT task. At a minimum, this should call `library( mvbutils)`, but you may also want to set some options controlling the behaviour of `cd` (see the **Options** section). If you use other features of `mvbutils` such as the function-editing interface in `fixr`, there will be further options to be set in `.First`. [MAC users: for some strange reason `.First` just doesn't get called if you are using the "usual" RGUI for MACs. So what you need to do is create a ".Rprofile" file in your ROOT folder using a text editor; this file should both contain the definition of the `.First` function, and should also call `.First()` directly. You can also put the `.First` commands directly into the ".Rprofile" file, but watch out for the side-effect of creating objects in `.GlobalEnv`.]

You can create a fully hierarchical structure, with subtasks within subtasks within tasks, etc. Even if your projects don't naturally look like this, you may find the facility useful. When I create a new task, I tend to start with just one level of hierarchy, containing data, function code, and results. When this gets unspeakably messy, I often create one (or more) subtasks, usually putting the basic data at the top level, and functions and results at the lower level. Apart from tidiness, this provides some degree of protection against overwriting the original data. And when even this gets too messy– in one task, I have more than 150 functions, and it is very easy to generate 100s of analysis results– I create another level, keeping "established" functions at the second tier and using the third tier for temporary workspace and results. There are no hard-and-fast rules here, of course, and different people use R in very different ways.

A task can have `.First.task` and/or `.Last.task` functions, which get called immediately after cding into the task from its parent, or immediately before cding back to its parent, respectively (see **Arguments**). These can be useful for dynamic loading, loading scripts into a text editor, attaching & detaching datasets, etc., and facilitate the use of tasks as informal packages.

For turning tasks into formal R packages, consult `mvbutils.packaging.tools`.

How it works

The mechanism underlying the tree structure is very simple: each task that has any subtasks will contain a character vector called `tasks`, whose names are the R names of the tasks, and whose elements are the corresponding disk directories. Your ROOT task need contain no more than a `.First` function and a `tasks` object.

You can manually modify the `tasks` vector, and sometimes this is essential. If you decide to move a disk directory, for example, you can manually change the corresponding element of `tasks` to reflect the change. (Though if you are moving a whole task hierarchy, e.g. when migrating to a new machine, consult `cd.change.all.paths`. Having said that, the ability to use relative pathnames in tasks, which is present since about mvbutils version 2.0, makes `cd.change.all.paths` partly redundant.) You can also rename a task very easily, via something like

```
names( tasks)[ names( tasks)=="my.old.name"] <- "my.new.name"
```

You can use similar methods to "reparent" a subtask without changing the directory structure.

There is (deliberately, to avoid accidents) no completely automatic way of removing tasks. To "hide" a task from the cd system, you first need to be cded to its parent; then remove the corresponding element of the tasks object, most easily via e.g.

```
tasks <- tasks %without.name% "mysubtask"
```

If you want to remove the directories corresponding to "mysubtask", you have to do so manually, either in the operating system or (for the brave) in R code.

Remember to Save() at some point after manually modifying tasks.

Options

Various options() can be set, as follows. Remember to put these into your .First function, too.

write.mvb.tasks=TRUE causes a sourceable text representation of the tasks object to be maintained in each directory, in the file tasks.r. This helps in case you accidentally wipe out the .RData file and lose track of where the child tasks live. To create these text representations for the first time throughout the hierarchy, call cd.write.mvb.tasks(0). You need to put the the options call in your .First.

abbreviate.cdprompt=n controls the length of the prompt string. Only the first n characters of all ancestral task names will be shown. For example, n=1 would replace the prompt long.task.name/data/funcs> with l/d/funcs>.

mvbutils.update.history.on.cd=FALSE will prevent automatic saving & reloading of the history file when cd is called.

cd checks the R_HISTFILE environment variable and, if unset, sets it to file.path(getwd()), ".Rhistory"). This (combined with the mvbutils replacement of the standard versions of savehistory and loadhistory– see package?mvbutils) ensures that the same history file is used throughout each and every R session. My experience is that a single master history file is safer. However, if you want to override this behaviour– e.g. if you want to use a separate history file for each task– call something like Sys.setenv(R_HISTFILE=".Rhistory") **before** the **first** use of cd.

Note

cd calls setwd so that file searches will default to the task directory (see also task.home).

cd always calls Save before attaching a child task on top or moving back up the hierarchy. If you have many and/or big objects, the default behaviour can be slow. You can speed this up– sometimes dramatically– by "mcacheing" some of your objects so that they are stored in separate files– see mlazy.

If there are no changes to the ".RData" file, cd will not modify the file– in particular, its date-of-access will be unchanged. This helps avoid unnecessary file copying on subsequent synchronization. However, there are several seemingly innocuous operations which change the workspace: calling a random number function (changes .Random.seed), causing an error (creates .Traceback), and causing a warning (creates last.warning). To avoid forcing a change to the entire ".RData" file whenever one of these changes, you can set option(mvbutils.quick.cd=TRUE); this turns on mcacheing for those objects (see mlazy), so that they are stored in separate mini-files.

cd is only meant to be called interactively, and has only been tested in that context.

cd will issue a warning and refuse to move back up the hierarchy if it detects a non-task attached in position 2. You will need to manually detach any such objects before cding back up, or write a .Last.task function to automatically do the detaching. To make sure that library (and any automatic loading of packages, e.g. if triggered by loading a file referring to a namespace) always inserts packages below ROOT, the .onLoad code in mvbutils makes a minor hack to library, changing the default pos argument accordingly.

Two objects in the mvb.session.info search environment (see search()) help keep track of what parts of the hierarchy are currently attached; .First.top.search and .Path. The former is set when mvbutils loads, and the latter is updated by cd. Attached tasks can be identified by having a path attribute consisting of a *named* character vector. Normal packages also have a path attribute, but without names.

Author(s)

Mark Bravington

See Also

move, task.home, cdtree, cdfind, cditerate, cd.change.all.paths, cd.write.mvb.tasks, cdprompt, fixr, mlazy

---

cdfind *Hierarchy-crawling functions for cd-organized workspaces*

Description

These functions work through part or all of a workspace (task) hierarchy set up via cd. cdfind searches for objects through the (attached and unattached) task hierarchy. cdtree displays the hierarchy structure. cd.change.all.paths is useful for moving or migrating all or part of the hierarchy to new disk directories. cd.write.mvb.tasks sets up sourceable text representations of the hierarchy, as a safeguard. cditerate is the engine that crawls through the hierarchy, underpinning the others; you can write your own functions to be called by cditerate.

If a task folder or its ".RData" file doesn't exist, a warning is given and (obviously) it's not iterated over. If that file does exist but there's a problem while loading it (e.g. a reference to the namespace of a package that can't be loaded– search for partial.namespaces in mvbutils.packaging.tools) then the iteration is still attempted, because something might be loaded. Neither case should cause an error.

Usage

```
cdfind( pattern, from = ., from.text, show.task.name=FALSE)
cdregexpr( regexp, from = ., from.text, ..., show.task.name=FALSE)
cdtree( from = ., from.text = substitute(from), charlim = 90)
cd.change.all.paths( from.text = "0", old.path, new.path)
cd.write.mvb.tasks( from = ., from.text = substitute(from))
cditerate( from.text, what.to.do, so.far = vector("NULL", 0), ..., show.task.name=FALSE)
## S3 method for class 'cdtree'
plot( x, ...) # S3 method for cdtree; normally plot( cdtree(<<args>>))
```

Arguments

pattern	regexpr to be checked against object names.
regexp	regexpr to be checked against function source code.
from	unquoted path specifier (see cd); make this 0 to operate on the entire hierarchy.
from.text	use this in place of from if you want to use a character string instead
show.task.name	(boolean) as-it-happens display of which task is being looked at
charlim	maximum characters per line allowed in graphical display of cdtree; reduce if unreadable, or change par(cex)
old.path	regexpr showing portion of directory names to be replaced
new.path	replacement portion of directory names
what.to.do	function to be called on each task (see **Details**)
so.far	starting value for accumulated list of function results
...	further fixed arguments to be passed to what.to.do (for cditerate), or grep (for cdregexpr), or foodweb (for plot.cdtree)
x	result of a call to cdtree, for plotting

Details

All these functions start by default from the task that is currently top of the search list, and only look further down the hiearchy (i.e. to unattached descendents). To make them work through the whole hierarchy, supply 0 as the from argument. cdtree has a plot method, useful for complicated task hierarchies.

If you want to automatically crawl through the task hierarchy to do something else, you can write a wrapper function which calls cditerate, and an inner function to be passed as the what.to.do argument to cditerate. The wrapper function will typically be very short; see the code of cdfind for an example.

The inner function (typically called cdsomething.guts) must have arguments found, task.dir, task.name, and env, and may have any other arguments, which will be set according as the ... argument of cditerate. found accumulates the results of previous calls to what.to.do. Your inner function can augment found, and should return the (possibly augmented) found. As for the other parameters: task.dir is obvious; task.name is a character(1) giving the full path specifier, e.g. "ROOT/mytask"; and env holds the environment into which the task has been (temporarily) loaded. env allows you to examine the task; for instance, you can check objects in the task by calling ls(env=env) inside your what.to.do function. See the code of cdfind.guts for an example.

Value

cdfind returns a list with one element for each object that is found somewhere; each such element is a character vector showing the tasks where the object was found. cdregexpr returns a list with one element for each task where a function whose source matches the regexpr is found; the names of each list element names the functions within that task (an ugly way to return results, for sure). cdtree returns an object of class cdtree, which is normally printed with indentations to show the hierarchy. You can also plot(cdtree(...)) to see a graphical display. cd.change.all.paths and cd.write.mvb.tasks do not return anything useful.

Author(s)

Mark Bravington

See Also

cd

Examples

```
cdfind( ".First", 0) # probably returns list( .First="ROOT")
```

---

cdprompt *Support routine for cd-organized workspace hierarchy.*

Description

Sets the command-line prompt to the correct value (see cd, and the notes on the option `abbreviate.cdprompt`); useful if the prompt somehow becomes corrupted. `cdprompt` never seems necessary in R but has been useful in the S+ manifestations of `mvbutils`, where system bugs are commoner.

Usage

```
cdprompt()
```

Author(s)

Mark Bravington

See Also

cd

Examples

```
cdprompt()
```

`changed.funs` *Show functions and callees in environment 'egood' that have changed or disappeared in environment 'ebad'.*

Description

Useful eg when you have been modifying a package, and have buggered stuff up, and want to partly go back to an earlier version... entirely hypothetical of course, things like that never ever happens to *me*. Mere mortals might want to create a new environment goodenv, use `evalq(source(<<old.mypack.R.source.file>> local=T), goodenv)`, then `find.changes( goodenv, asNamespace("mypack"))`. If your package is lazy-loaded, you're stuffed; I avoid lazy-loading, except perhaps for final distribution, because it just makes it much harder to track problems. Not that *I* ever have problems, of course.

Can be applied either to a specified set of functions, or by default to all the functions in egood. If the former, then all callees of the specified functions are also checked for changes, as are all their callees, and so on recursively.

Usage

```
changed.funs(egood, ebad, topfun, fw = NULL)
```

Arguments

`egood, ebad`	environments #1 & #2. Not symmetric; functions only in ebad won't be checked.
`topfun`	name of functions in egood to check; all callees will be checked too, recursively. Default is all functions in egood.
`fw`	if non-NULL, the result of a previous call to `foodweb(egood)`, but this will be called automatically if not.

Value

Character vector with the names of changed/lost functions.

`check.patch.versions` *Check consistency of maintained package versions*

Description

Utility to compare version numbers of the different "instances" of one of your maintained packages.
Only the most up-to-date folders relevant to the running R version are checked; see `mvbutils.packaging.tools`.

The "instances" checked are:

- the task package itself (in eg `..mypack$mypack.VERSION`)
- the source package created by `pre.install`
- the installed package, maintained by `patch.install`

- the tarball package, created by build.pkg
- the binary package, created by build.pkg.binary

The care argument controls what's shown. Mismatches when care="installed" should be addressed by patch.install, because something has gotten out-of-synch (probably when maintaining the same version of a package for different R versions). Mismatches with the built ("tarball" and "binary") packages are not necessarily a problem, just an indication of work-in-progress.

Usage

```
check.patch.versions(care = NULL)
```

Arguments

care	if non-NULL, a character vector with elements in the set "installed", "source", "tarball", and "binary". Only packages where there's a version mismatch between these fields and the task package version will be shown.

Value

A character matrix with maintained packages as rows, and the different instances as columns. "NA" indicates that a version couldn't be found.

---

ditto.list *Shorthand filler-inner for lists*

Description

Suppose you want to set up a list where several consecutive elements take the same value, but you don't want to repeatedly type that value: then use dittolist to set empty (missing) elements to the previous non-empty element. Wrap in unlist() to create a vector instead of a list.

Usage

```
ditto.list(...)
# EG:
# ditto.list( a=1, b=, c='hello') # a: 1; b: 1, c: 'hello'
```

Arguments

...	anything, named or unnamed; missing elements OK

Value

List

Examples

```
unlist( ditto.list( a=1, b=, c='hello')) # a: 1; b: 1, c: 'hello'
```

`do.in.envir` *Modify a function's scope*

Description

`do.in.envir` lets you write a function whose scope (enclosing environment) is defined at runtime, rather than by the environment in which it was defined.

Usage

```
# Use only as wrapper of function body, like this:
# my.fun <- function(...) do.in.envir( fbody, envir=)
# ... should be the arg list of "my.fun"
# fbody should be the code of "my.fun"
do.in.envir( fbody, envir=parent.frame(2)) # Don't use it like this!
```

Arguments

`fbody`	the code of the function, usually a braced expression
`envir`	the environment to become the function's enclosure

Details

By default, a `do.in.envir` function will have, as its enclosing environment, the environment in which it was **called**, rather than **defined**. It can therefore read variables in its caller's frame directly (i.e. without using `get`), and can assign to them via `<<-`. It's also possible to use `do.in.envir` to set a completely different enclosing environment; this is exemplified by some of the functions in `debug`, such as `go`.

Note the difference between `do.in.envir` and `mlocal`; `mlocal` functions evaluate in the frame of their caller (by default), whereas `do.in.envir` functions evaluate in their own frame, but have a non-standard enclosing environment defined by the `envir` argument.

Calls to e.g. `sys.nframe` won't work as expected inside `do.in.envir` functions. You need to offset the frame argument by 5, so that `sys.parent()` should be replaced by `sys.parent( 5)` and `sys.call` by `sys.call(-5)`.

`do.in.envir` functions are awkward inside namespaced packages, because the code in `fbody` will have "forgotten" its original environment when it is eventually executed. This means that objects in the namespace will not be found.

The **debug** package does not yet trace inside `do.in.envir` functions– this will change.

Value

Whatever `fbody` returns.

Author(s)

Mark Bravington

See Also

mlocal

Examples

```
fff <- function( abcdef) ffdie( 3)
ffdie <- function( x) do.in.envir( { x+abcdef} )
fff( 9) # 12; ffdie wouldn't know about abcdef without the do.in.envir call
# Show sys.call issues
# Note that the "envir" argument in this case makes the
# "do.in.envir" call completely superfluous!
ffe <- function(...) do.in.envir( envir=sys.frame( sys.nframe()), sys.call( -5))
ffe( 27, b=4) # ffe( 27, b=4)
```

---

do.on *Easier sapply/lapply avoiding explicit function*

Description

Simpler to demonstrate:

```
do.on( find.funs(), environment( get( .)))
# same as:
lapply( find.funs(), function( x) environment( get( x)))
```

do.on evaluates expr for all elements of x. The expression should involve the symbol ., and will be cast into a function which has an argument . and knows about any dotdotdot arguments passed to do.on (and objects in the function that calls do.on). If x is atomic (e.g. character or numeric, but not list) and lacks names, it will be given names via named. With do.on, you are calling sapply, so the result is simplified if possible, unless simplify=FALSE (or simplify="array", for which see sapply). With FOR, you are calling lapply, so no simplication is tried; this is often more useful for programming.

Usage

```
do.on(x, expr, ..., simplify = TRUE)
FOR(x, expr, ...)
```

Arguments

x	thing to be iterated over. Names are copied to the result, and are pre-allocated if required as per **Description**
expr	expression, presumably involving the symbol . which will successively become the individual elements of x
...	other "arguments" for expr
simplify	as per sapply, and defaulting to TRUE.

Value

do.on	as per sapply, a vector or array of the same "length" as x.
FOR	a list of the same length as x

Examples

```
do.on( 1:7, sum(1:.))
#  1  2  3  4  5  6  7
# 1  3  6 10 15 21 28
# note the numeric "names" in the first row
FOR( 1:3, sum(1:.))
```

---

doc2Rd *Converts plain-text documentation to Rd format*

Description

doc2Rd converts plain-text documentation into an Rd-format character vector, optionally writing it to a file. You probably won't need to call doc2Rd yourself, because pre.install and patch.install do it for you when you are building a package; the entire documentation of package **mvbutils** was produced this way. The main point of this helpfile is to describe plain-text documentation details. However, rather than wading through all the material below, just have a look at a couple of R's help screens in the pager, e.g. via help(glm, help_type="text"), copy the result into a text editor, and try making one yourself. Don't bother with indentation, except in item lists as per **More details** below (the pager's version is not 100% suitable). See fixr and its new.doc argument for how to set up an empty template: also help2flatdoc for how to convert existing Rd-format doco.

docotest allows you to quickly check what your plain-text doco would look life

Usage

```
doc2Rd( text, file=NULL, append=, warnings.on=TRUE, Rd.version=,
    def.valids=NULL, check.legality=TRUE)
docotest( fun.or.text, ...)
```

Arguments

	For doc2Rd:
	(character or function) character vector of documentation, or a function with a doc attribute that is a c.v. of d..
file	(string or connection) if non-NULL, write the output to this file
append	(logical) only applies if !is.null(file); should output be appended rather than overwriting?
warnings.on	(logical) ?display warnings about apparently informal documentation?

`Rd.version` (character) what Rdoc version to create "man" files in? Currently "1" means pre-R2.10, "2" means R2.10 and up. Default is set according to what version of R is running.

`def.valids` (character) objects or helpfiles for which links should be generated automatically. When `doc2Rd` is being called from `pre.install`, this will be set to all documented objects in your package. Cross-links to functions in other packages are not currently generated automatically (in fact not at all, yet).

`check.legality` if TRUE and `Rd.version` is 2 or more, then the output Rd will be run thru `parse_Rd` and a `try-error` will be returned if that fails; normal return otherwise. Not applicable if `Rd.version` is 1. For `docotest`:

`fun.or.text` (character or function) character vector of documentation, or a function with a `doc` attribute that is a c.v. of d.. NB if maintaining a package, you need to run this on the "raw" code (e.g. `..mypack$myfun`), not on the installed function (e.g. not `myfun` or `mypack::myfun`).

`...` other args passed to `Rd2HTML` when it tries to convert `doc2Rd` output to HTML. I've no idea what these might be, since they wouldn't be used in reality by `pre.install` when it assembles your source package.

Value

Character vector containing the text as it would appear in an Rd file, with `class` of "cat" so it prints nicely on the screen.

More details

Flat-format (plain-text) documentation in `doc` attributes, or in stand-alone character objects whose name ends with ".doc", can be displayed by the replacement `help` in `mvbutils` (see `dochelp`) without any further ado. This is very useful while developing code before the package-creation stage, and you can write such documentation any way you want. For display in an HTML browser (as opposed to R's internal pager), and/or when you want to generate a package, `doc2Rd` will convert pretty much anything into a legal Rd file. However, if you can follow a very few rules, using `doc2Rd` will actually give nice-looking authentic R help. For this to work, your documentation basically needs to look like a plain-text help file, as displayed by `help(..., help_type="text")`.

Rather than wading through this help file to work out how to write plain-text help, just have a look at a couple of R's help screens in the pager, and try making one yourself. You can also use `help2flatdoc` to convert an existing plain-text help file. Also check the file "sample.fun.rrr" in the "demostuff" subdirectory of this package (see **Examples**). If something doesn't work, delve more deeply...

- There are no "escape characters"– the system is "text WYSIWYG". For example, if you type a \ character in your doc, `help` will display a \ in that spot. Single quotes and percent signs can have special implications, though– see below.
- Section titles should either be fully capitalized, or end with a : character. The capitalized version shows up more clearly in informal help. Replace any spaces with periods, e.g. SEE.ALSO not SEE ALSO. The only non-alpha characters allowed are hyphens.
- Subsections are like sections, except they start with a sequence of full stops, one per nesting level. See also **Subsections**.

- "Item lists", such as in the **Arguments** section and sometimes the **Value** section (and sometimes other sections), should be indented and should have a colon to separate the item name from the item body.
- General lists of items, like this bullet-point list, should be indented and should start with a "-" character, followed by a space.
- Your spacing is generally ignored (exceptions: **Usage**, **Examples**, multi-line code blocks; see previous point). Tabs are converted to spaces. Text is wrapped, so you should write paragraphs as single lines without hard line breaks. Use blank lines generously, to make your life easier; also, they will help readability of informal helpfiles.
- To mark *in-line* code fragments (including variable names, package names, etc– basically things that R could parse), put them in single quotes. Hence you can't use single quotes within in-line code fragments.

An example of what you couldn't include:

```
'myfun( "'No no no!'")'
```

- Single quotes are OK within multi-code blocks, **Usage**, and **Examples**. For multi-line code blocks in other sections, don't bother with the single-quotes mechanism. Instead, insert a "%%#" line before the first line of the block, and make sure there is a blank line after the block.
- You can insert "hidden lines", starting with a % character, which get passed to the Rd conversion routines. If the line starts with %%, then the Rd conversion routines will ignore it too. The "%%#" line to introduce multi-line code blocks is a special case of this.
- Some other special constructs, such as links, can be obtained by using particular phrases in your documentation, as per **Special fields**.

Subsections: I've bolded some of these meta-refs to sections

Subsections are a nice new feature in R 2.11. You can use them to get better control over the order in which parts of documentation appear. R will order sections thus: **Usage**, **Arguments**, **Details**, **Value**, other sections you write in alphabetical section order, **Notes**, **See also**. That order is not always useful. You can add subsections to **Details** so that people will see them in the order you want. If you want **Value** to appear before **Details**, then just rename **Details** to "MORE.DETAILS", and put subsections inside that.

In plain-text, subsection headings are just like section headings, except they start with a period (don't use the initial periods when cross-referencing to it elsewhere in the doco). You can have nested subsections by adding extra periods at the start, like this:

Another depth of nesting: In the plain text version of this doco, the SUBSECTIONS line starts with one period, and the ANOTHER.DEPTH.OF.NESTING line starts with two. If you try to increase subsection depth by more than one level, i.e. with 2+ full stops more than the previous (sub)section, then `doc2Rd` will correct your "mistake".

Special fields: Almost anything between a pair of single quotes will be put into a \code{} or \code{\link{}} or \pkg{} or \env{} construct, and the quotes will be removed. A link will be used if the thing between the quotes is a one-word name of something documented in your package (assuming `doc2Rd` is being called from `pre.install`). A link will also be used in all cases of the form "See XXX" or "see XXX" or "XXX (qv)", where XXX is in single quotes, and any " (qv)" will be removed. With "[pP]ackage XXX" and "XXX package", a \pkg{} construct will be

used. References to .GlobalEnv and .BaseNamespaceEnv go into \env{} constructs. Otherwise, a \code{} construct will be used, unless the following exceptions apply. The first exception is if the quotes are inside **Usage**, **Examples**, or a multi-line code block. The second is if the first quote is preceded by anything other than " ", "(" or "-". The final semi-exception is that a few special cases are put into other constructs, as next.

URLs and email addresses should be enclosed in <...>; they are auto-detected and put into \url{} and \email{} constructs respectively.

Lines that start with a % will have the % removed before conversion, so their contents will be passed to RCMD Rdconv later (unless you start the line with %%). They aren't displayed by dochelp, though, so can be used to hide an unhelpful USAGE, say, or to hide an "#ifdef windows".

A solitary capital-R is converted to \R. Triple dots *used to be* converted to \dots (regardless of whether they're in code or normal text) but I've stopped doing so because this conversion was taking 97% of the total runtime!

Any reasonable "*b*old" or "_emphatic stuff_" constructions (no quotes, just the asterisks) will go into \bold{} and \emph{} constructs respectively, to give **b**old or *emphatic stuff*. (Those first two didn't, because they are "unreasonable"– in particular, they're quoted.) No other fancy constructs are supported (yet).

Format for non-function help: For documenting datasets, the mandatory sections seem to be **Description**, **Usage**, and **Format**; the latter works just like **Arguments**, in that you specify field names in a list. Other common sections include **Examples**, **Source**, **References**, and **Details**.

Extreme details: The first line should be the docfile name (without the Rd) followed by a few spaces and the package descriptor, like so:

utility-funs package:mypack

When doc2Rd runs, the docfile name will appear in both the \name{} field and the first \alias{} field. pre.install will actually create the file "utility-funs.Rd". The next non-blank lines form the other alias entries. Each of those lines should consist of one word, preceded by one or more spaces for safety (not necessary if they have normal names).

"Informal documentation" is interpreted as any documentation that doesn't include a "DESCRIPTION" (or "Description:") line. If this is the case, doc2Rd first looks for a blank line, treats everything before it as \alias{} entries, and then generates the **Description** section into which all the rest of your documentation goes. No other sections in your documentation are recognized, but all the special field substitutions above are applied. (If you really don't want them to be, use the multi-line code block mechanism.) Token **Usage**, **Arguments**, and **Keywords** sections are appended automatically, to keep RCMD happy.

Section titles built into Rd are: **Description**, **Usage**, **Synopsis** (defunct for R>=3.1), **Arguments**, **Value**, **Details**, **Examples**, **Author** or **Author(s)**, **See also**, **References**, **Note**, **Keywords** and, for data documentation only, **Format** and **Source**. Other section titles (in capitals, or terminated with a colon) can be used, and will be sentence-cased and wrapped in a \section{} construct. Subsections work like sections, but begin with a sequence of full stops, one per nesting level. Most cross-refs to (sub)sections will be picked up automatically and put into **bold**, so that e.g. "see MY.SECTION" will appear as "see **My section**"; when referring to subsections, omit the initial dots. To force a cross-reference that just doesn't want to appear, use e.g. "MY.SECTION (qv)", or just wrap it in "*...*".

The \docType field is set automatically for data documentation (iff a **Format** section is found) and for package documentation (iff the name on the first line includes "-package").

Spacing within lines does matter in **Usage** (qv), **Examples**, and multi-line code blocks, where what you type really is what you get (except that a fixed indent at the start of all lines in such a block is removed, usually to be reinstated later by the help facilities). The main issue is in the package "manual" that RCMD generates for you, where the line lengths are very short and overflows are common. (Overflows are also common with in-line code fragments, but little can be done about that.) The "RCMD Rd2dvi –pdf" utility is helpful for seeing how individual helpfiles come out.

In **See also**, the syntax is slightly different; names of things to link to should *not* be in single quotes, and should be separated by commas or semicolons; they will be put into \code{\link{}} constructs. You can split SEE.ALSO across several lines; this won't matter for pager help, but can help produce tidier output in the file "***-manual.tex" produced by RCMD CHECK.

In **Examples**, to designate "don't run" segments, put a "## Don't run" line before and a "## End don't run" line after.

I never bother with **Keywords**, but if you do, then separate the keywords with commas, semicolons, or line breaks; don't use quotes. A token **Keywords** section will be auto-generated if you don't include one, to keep RCMD happy.

Infrequently asked questions: **Q:** Why didn't you use Markdown/MyPetBargainSyntax?

A: Mainly because I didn't know about them, to be honest. But WRTO MarkDown it seemed to me that the hard-line-breaks feature would be a pain. If anyone thinks there's really good alternative standard, please let me know.

Q: I have written a fancy *displayed* equation using \deqn{} and desperately want to include it. Can I?

A: Yes (though are you sure that a fancy equation really belongs in your function doco? how about in an attached PDF, or vignette?). Just prefix all the lines of your \deqn with %. If you want something to show up in informal help too, then make sure you also include lines with the text version of the equation, as per the next-but-one question.

Q: I have written a fancy *in-line* equation using \eqn{} and desperately want to include it. Can I?

A: No. Sorry.

Q: For some reason I want to see one thing in informal help (i.e. when the package isn't actually loaded but just sitting in a task on the search path), but a different thing in formal help. Can I do that?

A: If you must. Use the %-line mechanism for the formal help version, and then insert a line "%#ifdef flub" before the informal version, and a line "%#endif" after it. Your text version will show up in informal help, and your fancy version will show up in all help produced via Rd. (Anyone using the "flub" operating system will see both versions...)

Q: How can I insert a file/kbd/samp/option/acronym etc tag?

A: You can't. They all look like single quotes in pager-style help, anyway.

Q: What about S3?

A: S3 methods often don't need to be documented. However, they can be documented just like any other function, except for one small detail: in the **Usage** section, the call should use the generic name instead of your method name, and should be followed by a comment "# S3 method for <class>"; you can append more text to the comment if you wish. E.G.: if you are documenting a method `print.cat`, the **Usage** section should contain a call to `print(x,...) # S3 method for cat` rather than `print.cat(x,...)`. The version seen by the user will duplicate this "S3 method..." information, but never mind eh.

If you are also (re)defining an S3 generic and documenting it in the same file as various methods, then put a comment `# generic` on the relevant usage line. See `?print.function` for associated requirements.

Confusion will deservedly arise with a function that looks like an S3 method, but isn't. It will be not be labelled as S3 by `pre.install` because you will of course have used the full name in the **Usage** section, because it isn't a method. However, it can still be found by `NextMethod` etc., so you shouldn't do that. (Though `mvbutils::max.pkg.ver` currently does exactly that...)

S3 classes themselves need to be documented either via a relevant method using an alias line, or via a separate `myclass.doc` text object.

Q: What about S4?

A: I am not a fan of S4 and have found no need for it in many 1000s of lines of R code... hence I haven't included any explicit support for it so far. Nevertheless, things might well work anyway, unless special Rd constructs are needed. If `doc2Rd` *doesn't* work for your S4 stuff (bear in mind that the %-line mechanism may help), then for now you'll still have to write S4 Rd files yourself; see `pre.install` for where to put them. However, if anyone would like the flatdoc facility for S4 and is willing to help out, I'm happy to try to add support.

See Also

The file "sample.fun.rrr" in subdirectory "demostuff", and the demo "flatdoc.demo.r".

To do a whole group at once: `pre.install`.

To check the results: `docotest(myfun)` to check the HTML (or `patch.installed(mypack)` and then `?myfun`). TODO something to easily check PDF (though R's PDF doco is pointless IMO); for now you need to manually generate the file, then from a command-line prompt do something like "RCMD Rd2dvi –pdf XXX.Rd" and "RCMD Rdconv -t=html XXX.Rd" and/or "-t=txt"

To convert existing Rd documentation: `help2flatdoc`.

If you want to tinker with the underlying mechanisms: `flatdoc`, `write.sourceable.function`

Examples

```
## Needs a function with the right kind of "doc" attr
## Look at file "demostuff/sample.fun.rrr"
sample.fun <- source.mvb( system.file( file.path(
    'demostuff', 'sample.fun.rrr'), package='mvbutils'))
print( names( attributes( sample.fun)))
cat( '***Original plain-text doco:***\n')
print( as.cat( attr( sample.fun, 'doc'))) # unescaped, ie what you'd actually edit
cat( '\n***Rd output:***\n')
sample.fun.Rd <- doc2Rd( sample.fun)
print( sample.fun.Rd) # already "cat" class
## Not run:
docotest( sample.fun) # should display in browser

## End(Not run)
```

dochelp *Documentation (informal help)*

Description

dochelp(topic) will be invoked by the replacement help if conventional help fails to find documentation for topic topic. If topic is an object with a doc attribute (or failing that if <<topic>> or <<topic>>.doc is a character vector), then the attribute (or the character object) will be formatted and displayed by the pager or browser. dochelp is not usually called directly.

Usage

```
# Not usually called directly
# If it is, then normal usage is: dochelp( topic)
dochelp( topic, doc, help_type=c( "text", "html"))
# Set options( mvb_help_type="text") if the browser gives you grief
```

Arguments

topic	(character) name of the object to look for help on, or name of "...doc" character object– e.g. either thing or thing.doc if the character object is thing.doc.
doc	(character or list)– normally not set, but deduced by default from topic; see **Details**.
help_type	as per help. Defaults to getOption("mvb_help_type") in normal usage, which in turn defaults to getOption("help_type") as for standard help. Only "text" and "html" are supported by dochelp; anything else maps to "text", which invokes R's internal pager.

Details

dochelp will only be called if the original help call was a simple help(topic=X, ...) form, with X not a call and with no try.all.packages or type or lib.loc arguments (the other help options are OK).

The doc argument defaults to the doc attribute of get("topic"). The only reason to supply a non-default argument would be to use dochelp as a pager; this might have some value, since dochelp does reformat character vectors to fit nicely in the system pager window, one paragraph per element, using strwrap. Elements starting with a "%" symbol are not displayed.

To work with dochelp, a doc attribute should be either:

- a character vector, of length >=1. New elements get line breaks in the pager. Or:
- a length-one list, containing the name of another object with a doc attribute. dochelp will then use the doc attribute of that object instead. This referencing can be iterated.

If the documentation is very informal, start it with a blank line to prevent find.documented(..., doctype="Rd") from finding it.

With help_type="text", the doco will be re-formatted to fit the pager; each paragraph should be a single element in the character vector. Elements starting with a % will be dropped (but may still be useful for doc2Rd).

With help_type="html", the doco will be passed thru doc2Rd and then turned into HTML. doc2Rd is pretty forgiving and has a fair crack at converting even very informal documentation, but does have its limits. If there is an error in the doc2Rd conversion then help_type will be reset to "text".

flatdoc offers an easy way to incorporate plain-text (flat-format) documentation– formal or informal– in the same text file as a function definition, allowing easy maintenance. The closer you get to the displayed appearance of formal R-style help, the nicer the results will look in a browser (assuming help_type="html"), but the main thing is to just write *some* documentation– the perfect is the enemy of the good in this case!

Author(s)

Mark Bravington

See Also

flatdoc, doc2Rd, find.documented, strwrap

Examples

```
#
myfun <- structure( function() 1,
  doc="Here is some informal documentation for myfun\n")
dochelp( "myfun")
help( "myfun") # calls dochelp
```

---

dont.lock.me *Prevent sealing of a namespace, to facilitate package maintenance.*

Description

Call dont.lock.me() during a .onLoad to stop the namespace from being sealed. This will allow you to add/remove objects to/from the namespace later in the R session (in a sealed namespace, you can only change objects, and you can't unseal a namespace retrospectively). There could be all sorts of unpleasant side-effects. Best to leave it to maintain.packages to look after this for you...

Usage

```
# default of env works if called directly in .onLoad
dont.lock.me( env=environment( sys.function( -1)))
```

Arguments

env	the environment to not lock.

Details

dont.lock.me hacks the standard lockEnvironment function so that locking won't happen if the environment has a non-NULL dont.lock.me attribute. Then it sets this attribute for the namespace environment.

Examples

```
## Not run:
# This unseals the namespace of MYPACK only if the option "maintaining.MYPACK" is non-NULL:
.onLoad <- function( libname, pkgname) {
  if( !is.null( getOption( 'maintaining.' %&% pkgname)))
    mvbutils:::dont.lock.me()
}

## End(Not run)
```

---

dont.lockBindings *Helper for live-editing of packages*

Description

Normally, objects in a NAMESPACEd package are locked and can't be changed. Sometimes this isn't what you want; you can prevent it by calling dont.lockBindings in the .onLoad for the package. For user-visible objects (i.e. things that end up in the "package:blah" environment on the search path), you can achieve the same effect by calling dont.lockBindings in the package's .onAttach function, with namespace=FALSE.

Usage

```
dont.lockBindings( what, pkgname, namespace.=TRUE)
```

Arguments

what	(character) the names of the objects to not lock.
pkgname	(string) the name of the package. As you will only use this inside .onLoad, you can just set this to pkgname which is an argument of .onLoad.
namespace.	TRUE to antilock in the namespace during .onLoad; FALSE to antilock in the visible manifestation of the package.

Details

Locking occurs after .onLoad / .onAttach are called so, to circumvent it, dont.lockBindings creates a hook function to be called after the locking step.

See Also

lockBinding, setHook

Examples

```
## Not run:
library( debug)
debug:::.onLoad # d.lB is called to make 'tracees' editable inside 'debug's namespace.
debug:::.onAttach # d.lB is called to make 'tracees' editable in the search path
# NB also that an active binding is used to ensure that the 'tracees' object in the search...
#... path is a "shadow of" or "pointer to" the one in 'debug's namespace; the two cannot get...
#... out-of-synch

## End(Not run)
```

`extract.named` *Create variables from corresponding named list elements*

Description

This is a convenience function for creating named variables from lists. It's particularly useful for "unpacking" the results of calls to `.C`.

Usage

```
extract.named( l, to=parent.frame())
```

Arguments

`l`	a list, with some named elements (no named elements is OK but pointless)
`to`	environment

Value

nothing directly, but will create variables

Author(s)

Mark Bravington

Examples

```
ff <- function(...) { extract.named( list(...)); print( ls()); bbb }
# note bbb is not "declared"
ff( bbb=6, ccc=9) # prints [1] "bbb" "ccc", returns 6
```

fast.read.fwf *Read in fixed-width files quickly*

Description

Experimental replacement for `read.fwf` that runs much faster. Included in `mvbutils` only to reduce dependencies amongst my other packages.

Usage

```
fast.read.fwf(file, width,
    col.names = if (!is.null(colClasses))
        names( colClasses) else "V" %&%  1:ncol(fields),
    colClasses = character(0), na.strings = character(0L),  tz = "", ...)
```

Arguments

`file`	character
`width`	vector of column widths. Negative numbers mean "skip this many columns". Use an NA as the final element if there are likely to be extra characters at the end of each row after the last one that you're interested in.
`col.names`	names for the columns that are NOT skipped
`colClasses`	can be used to control type conversion; see `read.table`. It is an optional vector whose names must be part of `col.names`. There is one extension of the `read.table` rules:a `colClass` string starting `POSIXct.` will trigger automatic conversion to `POSIXct`, using the rest of the string as the format specifier. See also `tz`.
`na.strings`	are there any strings (other than NA) which should convert to NAs?
`tz`	used in auto-conversion to `POSIXct` when `colClass` is set
`...`	ignored; it's here so that this function can be called just like `read.fwf`

Value

A `data.frame`, as per `read.fwf` and `read.table`. misc

find.documented *Support for flat-format documentation*

Description

`find.documented` locates functions that have flat-format documentation; the functions and their documentation can be separate, and are looked for in all the environments in `pos`, so that functions documented in one environment but existing in another will be found. `find.docholder` says where the documentation for one or more functions is actually stored. Both `find.documented` and `find.docholder` check two types of object for documentation: (i) functions with "doc" attributes, and (ii) character-mode objects whose name ends in ".doc"

Usage

```
find.documented( pos=1, doctype=c( "Rd", "casual", "own", "any"),
  only.real.objects=TRUE)
find.docholder( what, pos=find( what[1]))
```

Arguments

pos	search path position(s), numeric or character. In find.documented, any length. In find.docholder, only pos[1] will be used; it defaults to where the first element of what is found.
doctype	Defaults to "Rd". If supplied, it is partially matched against the choices in **Usage**. "Rd" functions are named in the alias list at the start of (i) any doc attribute of a function, and (ii) any text object whose name ends with ".doc", that exist in pos (see doc2Rd). "casual" functions have their own doc attribute and will be found by the replacement of help; note that the doc attribute can be just a reference to another documented function, of mode "list" as described in dochelp. "own" functions (a subset of "casual") have their own character-mode doc attribute, and are suitable for doc2Rd. "any" combines casual and Rd.
only.real.objects	If TRUE, only return names of things that exist somewhere in the pos environments. FALSE means that other things such as the name of helpfiles might be returned, too.
what	names of objects whose documentation you're trying to find.

Value

find.documented	Character vector of function names.
find.docholder	list whose names are what; element i is a character vector showing which objects hold documentation for what[i]. Normally you'd expect either 0 or 1 entries in the character vector; more than 1 would imply duplication.

Note

doctype="Rd" looks for the alias names, i.e. the first word of all lines occurring before the first blank line. This may include non-existent objects, but these are checked for and removed.

Start informal documentation (i.e. not intended for doc2Rd) with a blank line to avoid confusion.

Author(s)

Mark Bravington

See Also

flatdoc, doc2Rd, dochelp

`fix.order` *Shows functions sorted by date of edit*

Description

`fix.order` sorts the functions according to the filedates of their backups (in the .Backup.mvb directory). This is very useful for reminding yourself what you were working on recently. It only works if functions have been edited using the `fixr` system.

Usage

```
fix.order( env=1)
```

Arguments

`env`	a single number, character string, or environment. Numbers and characters are interpreted as search path positions. The environment must be an attached mvb-style task.

Details

Only functions that have a BU*** backup file will appear. Functions that have a BU*** file but have been deleted will not appear.

Value

Character vector of functions sorted by date/time of last modification.

To do

Probably should modify this so it takes an arbitrary task path instead of a search position only. Task doesn't really need to be attached.

Add a `pattern` argument a la find.funs.

Author(s)

Mark Bravington

See Also

`fixr`

Examples

```
## Not run:
## Need to create backups and do some function editing first
fix.order() # functions in .GlobalEnv
fix.order( "ROOT") # functions in your startup task

## End(Not run)
```

---

fixr *Editing functions, text objects, and scriptlets*

Description

`fixr` opens a function (or text object, or "script" stored as an R `expression`— see **Scriptlets**) in your preferred text editor. Control returns immediately to the R command line, so you can keep working in R and can be editing several objects simultaneously (cf `edit`). A session-duration list of objects being edited is maintained, so that each object can be easily sourced back into its rightful workspace. These objects will be updated automatically on file-change if you've run `autoedit( TRUE)` (e.g. in your `.First`), or manually by calling `FF()`. There is an optional automatic text backup facility.

The safest is to call `fixtext` to edit text objects, and `fixr` for functions and everything else. However, `fixr` can handle both, and for objects that already exist it will preserve the type. For new objects, though, you have to specify the type by calling either `fixr` or `fixtext`. If you forget— ie if you really wanted to create a new text object, but instead accidentally typed `fixr( mytext)`— you will (probably) get a parse error, and `mytext` will then be "stuck" as a broken function. Your best bet is to copy the actual contents in the text-editor to the clipboard, type `fixtext( mytext)` in R, paste the old contents into the text-editor, and save the file; R will then reset the type and all should be well.

`readr` also opens a file in your text editor, but in read-only mode, and doesn't update the backups or the list of objects being edited.

Usage

```
# Usually: fixr( x) or fixr( x, new.doc=T)
fixr( x, new=FALSE, install=FALSE, what, fixing, pkg=NULL,
    character.only=FALSE, new.doc=FALSE, force.srcref=FALSE)
# fixtext really has exact same args as fixr, but technically its args are:
fixtext( x, ...)
# Usually: readr( x) but exact same args as fixr, though the defaults are different
readr( x, ...)
FF() # manual check and update, usually only needed...
    # ... temporarily if autoedit() stops working
autoedit( do=TRUE) # stick this line in your .First
```

Arguments

x	a quoted or unquoted name of a function, text object, or expression. You can also write mypack$myfun, or mypack::myfun, or mypack:::myfun, or ..mypack$myfun, to simultaneously set the pkg argument (only if mypack has been set up with maintain.packages). Note that fixr uses non-standard evaluation of its x argument, unless you specify character.only=TRUE. If your object has a funny name, either quote it and set character.only=TRUE, or pass it directly as...
character.only	(logical or character) if TRUE, x is treated as a string naming the object to be edited, rather than the unquoted object name. If character.only is a string, it is treated as the name of x, so that eg fixr(char="funny%name") works.
new.doc	(logical) if TRUE, add skeleton plain-text R-style documentatation, as per add.flatdoc.to. Also use this to create an empty scriptlet for a general (non-function, non-text) object.
force.srcref	(logical) Occasionally there have been problems transferring old code into "new" R, especially when a function has text attributes such as (but not limited to) doc; the symptom is, they appear in the editor just as "# FLAT-FORMAT DOCUMENTATION". This sometimes requires manual poking-around, but usually can be sorted out by calling fixr(...,force.srcref=TRUE).
new	(logical, seldom used) if TRUE, edit a blank function template, rather than any existing object of that name elsewhere in the search path. New edit will go into .GlobalEnv unless argument pkg is set.
install	(logical, rarely used) logical indicating whether to go through the process of asking you about your editor
what	Don't use this– it's "internal"! [Used by fixtext, which calls fixr with what="" to force text-mode object. what should be an object with the desired class.]
fixing	(logical, rarely used) FALSE for read-only (i.e. just opening editor to examine the object)
pkg	(string or environment) if non-NULL, then specifies in which package a specific maintained package (see maintain.packages) x should be looked for.
do	(logical) TRUE => automatically update objects from altered files; FALSE => don't.
...	other arguments, except what in fixtext, and fixing in readr, are passed to fixr.

Details

When fixr is run for the first time (or if you set install=TRUE), it will ask you for some basic information about your text editor. In particular, you'll need to know what to type at a command prompt to invoke your text editor on a specific file; in Windows, you can usually find this by copying the Properties/Shortcut/Target field of a shortcut, followed by a space and the filename. After supplying these details, fixr will launch the editor and print a message showing some options ("backup.fix", "edit.scratchdir" and "program.editor"), that will need to be set in your .First. function. You should now be able to do that via fixr(.First).

Changes to the temporary files used for editing can be checked for automatically whenever a valid R command is typed (e.g. by typing 0<ENTER>; <ENTER> alone doesn't work). To set this up, call `autoedit()` once per session, e.g. in your `.First`. The manual version (ie what `autoedit` causes to run automatically) is `FF()`. If any file changes are detected by `FF`, the code is sourced back in and the appropriate function(s) are modified. `FF` tries to write functions back into the workspace they came from, which might not be `.GlobalEnv`. If not, you'll be asked whether you want to `Save` that workspace (provided it's a task– see `cd`). `FF` should still put the function in the right place, even if you've called `cd` after calling `fixr` (unless you've detached the original task) or if you `move`d it. If the function was being `mtrace`d (see `package?debug`), `FF` will re-apply `mtrace` after loading the edited version. If there is a problem with parsing, the `source` attribute of the function is updated to the new code, but the function body is invisibly replaced with a `stop` call, stating that parsing failed.

If something goes wrong during an automatic call to `FF`, the automatic-call feature will stop working; this is rare, but can be caused eg by hitting <ESC> while being prompted whether to save a task. To restart the feature in the current R session, do `autoedit(F)` and then `autoedit(T)`. It will come back anyway in a new R session.

`readr` requires a similar installation process. To get the read-only feature, you'll need to add some kind of option/switch on the command line that invokes your text editor in read-only mode; not all text editors support this. Similarly to `fixr`, you'll need to set `options( program.reader=<<something>>)` in your `.First`; the installation process will tell you what to use.

`fixr`, and of course `fixtext`, will also edit character vectors. If the object to be edited exists beforehand and has a class attribute, `fixr` will not change its class; otherwise, the class will be set to "cat". This means that `print` invokes the `print.cat` method, which displays text more readably than the default. Any other attributes on character vectors are stripped.

For functions, the file passed to the editor will have a ".r" extension. For character vectors or other things, the default extension is ".txt", which may not suit you since some editors decide syntax-highlighting based on the file extension. (EG if the object is a character-vector "R script", you might want R-style syntax highlighting.) You can somewhat control that behaviour by setting `options()$fixr.suffices`, eg

```
options( fixr.suffices=c( r='.r', data='.dat'))
```

which will mean that non-function objects whose name ends `.r` get written to files ending ".r.r", and objects whose name ends `.data` get written to files ending ".data.dat"; any other non-functions will go to files ending ".txt". This does require you to use some discipline in naming objects, which is no bad thing; FWIW my "scripts" always do have names ending in `.r`, so that I can see what's what.

`fixr` creates a blank function template if the object doesn't exist already, or if `new=TRUE`. If you want to create a new character vector as opposed to a new function, call `fixtext`, or equivalently set `what=""` when you call `fixr`.

If the function has attributes, the version in the text editor will be wrapped in a `structure(...)` construct (and you can do this yourself). If a `doc` attribute exists, it's printed as free-form text at the end of the file, and the call to `structure` will end with a line similar to:

```
,doc=flatdoc( EOF="<<end of doc>>"))
```

When the file is sourced back in, that line will cause the rest of the file– which should be free-format text, with no escape characters etc.– to be read in as a doc attribute, which can be displayed by help. If you want to add plain-text documentation, you can also add these lines yourself– see flatdoc. Calling fixr(myfun, new.doc=TRUE) sets up a documentation template that you can fill in, ready for later conversion to Rd format in a package (see mvbutils.packaging.tools).

The list of functions being edited by fixr is stored in the variable fix.list in the mvb.session.info environment. When you quit and restart R, the function files you have been using will stay open in the editor, but fix.list will be empty; hence, updating the file "myfun.r" will not update the corresponding R function. If this happens, just type fixr(myfun) in R and when your editor asks you if you want to replace the on-screen version, say no. Save the file again (some editors require a token modification, such as space-then-delete, first) and R will notice the update. Very very occasionally, you may want to tell R to stop trying to update one of the things it's editing, via eg fixtext <<- fixtext[-3,] if the offending thing is the third row in fixlist; note the double arrow.

An automatic text backup facility is available from fixr: see ?get.backup. The backup system also allows you to sort edited objects by edit date; see ?fix.order.

Changes with r 2 14: Time was, functions had their source code (including comments, author's preferred layout, etc) stored in a "source" attribute, a simple character vector that was automatically printed when you looked at the function. Thanks to the fiddly, convoluted, opaque "srcref" system that has replaced "source" as of R 2.14— to no real benefit that I can discern— fixr in versions of mvbutils prior to 2.5.209 didn't work correctly with R 2.14 up. Versions of mvbutils after 2.5.509 should work seamlessly.

The technical point is that, from R 2.14 onwards, basic R will *not* show the source attribute when you type a function name without running the function; unless there is a srcref attribute, all you will see is the deparsed raw code. Not nice; so the replacement to print.function in mvbutils will show the source attribute if it and no srcref attribute is present. As soon as you change a function with fixr post-R-2.14, it automatically loses any source attribute and acquires a "proper" srcref attribute, which will from then on.

Local function groups: There are several ways to work with "nested" (or "child" or "lisp-style macro") functions in R, thanks to R's scoping and environment rules; I've used at least four, most often mlocal in package **mvbutils**. One is to keep a bunch of functions together in a local environment so that they (i) know about each other's existence and can access a shared variable pool, (ii) can be edited en bloc, but (iii) don't need to clutter up the "parent" code with the definitions of the children. fixr will happily create & edit such a function-group, as long as you make sure the last statement in local evaluates to a function. For example:

```
# after typing 'fixr( secondfun)' in R, put this into your text editor:
local({
  tot <- 0
  firstfun <- function( i) tot <<- tot+i
  function( j) {
      for( ii in 1:j)
        firstfun( ii)
      tot
    }
})
```

Note that it's *not* necessary to assign the last definition to a variable inside the local call, unless you want to be able to reach that function recursively from one of the others, as in the first example

for local. Note also that firstfun will not be visible "globally", only from within secondfun when it executes.

secondfun above can be debugged as usual with mtrace in the **debug** package. If you want to turn on mtracing for firstfun without first mtracing secondfun and manually calling mtrace(firstfun) when secondfun appears, do mtrace(firstfun, from=environment(secondfun)).

Note: I *think* all this works OK in normal use (Oct 2012), but be careful! I doubt it works when building a package, and I'm not sure that R-core intend that it should; you might have to put the local-building code into the .onLoad.

Scriptlets: **Note**: I've really gone off "scriptlets" (writing this in mid 2016). These days I prefer to keep "scripts" as R character-vector objects (because I dislike having lots of separate files), edited by fixtext and manually executed as required by debug::mrun— which also has a debugging option that automatically applies mtrace. I'm not going to remove support for scriptlets in fixr, but I'm not going to try hard to sort out any bugs either. Instructions below are unchanged, and unchecked, from some years ago.

You can also maintain "scriptlets" with fixr, by embedding the instructions (and comments etc) in an expression(...) statement. Obviously, the result will be an expression; to actually execute a scriptlet after editing it, use eval(). The scriptlet itself is stored in the "source" attribute as a character vector of class cat, and the expression itself is given class thing.with.source so that the source is displayed in preference to the raw expression. Backup files are maintained just as for functions. Only the *first* syntactically complete statement is returned by fixr (though subsequent material, including extra comments, is always retained in the source attribute); make sure you wrap everything you want done inside that call to expression(...).

Two cases I find useful are:

- instructions to create data.frames or matrices by reading from a text file, and maybe doing some initial processing;
- expressions for complicated calls with particular datasets to model-fitting functions such as glm.

```
# Object creator:
expression( { # Brace needed for multiple steps
  raw.data <- read.table( "bigfile.txt", header=TRUE, row=NULL)
  # Condense date/time char fields into something more useful:
  raw.data <- within( raw.data, {
   Time <- strptime( paste( DATE, TIME, sep=' '), format="%Y-%m-%d %H:%M:%S")
    rm( DATE, TIME)
  })
  cat( "'raw.data' created OK")
})
```

and

```
# Complicated call:
expression(
 glm( LHS ~ captain + beard %in% soup, data=alldata %where% (mushroom=='magic'), family=binomial( li
)
```

Bear in mind that eval(myscriptlet) takes place in .GlobalEnv unless you tell it not to, so the first example above actually creates raw.data even though it returns NULL. To trace evaluation of myscriptlet with the **debug** package, call debug.eval(myscriptlet).

For a new scriptlet `mything`, the call to `fixr` should still just be `fixr(mything)`. However, if you have trouble with this, try `fixr( mything, what=list())` instead, even if `mything` won't be a `list()`. For an existing non-function, you'll need the `new=T` argument, e.g. `fixr( oldthing, new=T)`, and you'll then have to manually copy/paste the contents.

Note that you **can't** use `quote()` instead of `expression()`, because any attempt to display the object will cause it to run instead; this is a quirk of S3 methods!

For the brave: In principle, you can also edit non-expressions the same way. For example, you can create a `list` directly (not requiring subsequent `eval()`) via a scriptlet like this:

```
list(
  a = 1, # a number
  b = 'aardvark' # a character
)
```

Nowadays I tend to avoid this, because the code will be executed immediately R detects a changed file, and you have no other (easy) control over when it's evaluated. Also, note that the result will have class `thing.with.source` (prepended to any other S3 classes it might have), which has its own print method that shows the source; hence you won't see the contents directly when you just type its name, which may or may not be desirable.

Troubleshooting

Rarely, `fixr` (actually `FF`) can get confused, and starts returning errors when trying to update objects from their source files. (Switching between "types" of object with the same name— function, expression, character vector— can do this.) In such cases, it can be useful to purge the object from the `fix.list`, a session-duration data.frame object in workspace `mvb.session.info` on the search path. Say you are having trouble with object "badthing": then

```
fix.list <<- fix.list[ names( fix.list) != 'bad.thing',]
```

will do the trick (note the double arrow). This means `FF` will no longer look for updates to the source file for `badthing`, and you are free to again `fixr( badthing)`.

To purge the entire `fix.list`, do this:

fix.list «- fix.list[0,]

Note

`fixr` is designed to be used with `cd`; I'm not sure it will work independently.

Originally, `fixr` was only for functions, and not even for functions in packages, so that it was mostly an alternative to e.g. ESS; if you liked ESS, you wouldn't have bothered with `fixr`. However, `fixr` now has more sophisticated purposes, in particular being AFAIK the only reliable way of interfacing the package-maintenance features in the **mvbutils** package. It would be interesting to find out if it can be integrated with e.g. ESS (which I know only enough about to dislike). Input welcome (but unexpected; none has ever come from ESSers).

See Also

`.First`, `edit`, `cd`, `get.backup`, `fix.order`, `move`

flatdoc *Flat-format documentation*

Description

The flatdoc convention lets you edit plain-text documentation in the same file as your function's source code. flatdoc is hardly ever called explicitly, but you will see it in text files produced by fixr; you can also add it to such files yourself. mvbutils extends help so that ?myfunc will display this type of documentation for myfunc, even if myfunc isn't in a package. There are no restrictions on the format of informal-help documentation, so flatdoc is useful for adding quick simple help just for you or for colleagues. If your function is to be part of a maintained package (see mvbutils.packaging.tools), then the documentation should follow a slightly more formal structure; use fixr(myfun, new.doc=T) to set up the appropriate template.

Usage

```
# ALWAYS use it like this:
# structure( function( ...) {body},
# doc=flatdoc( EOF="<<end of doc>>"))
# plaintext doco goes here...
# NEVER use it like this:
flatdoc( EOF="<<end of doc>>")
```

Arguments

EOF	character string showing when plain text ends, as in readlines.mvb
body	replace with your function code
...	replace with your function arg list

Value

Character vector of class docattr, as read from the current.source() (qv) connection. The print method for docattr objects just displays the string "# FLAT-FORMAT DOCUMENTATION", to avoid screen clutter.

Internal details

This section can be safely ignored by almost all users.

On some text editors, you can modify syntax highlighting so that the "start of comment block" marker is set to the string "doc=flatdoc(".

It's possible to use flatdoc to read in more than one free-format text attribute. The EOF argument can be used to distinguish one block of free text from the next. These attributes can be accessed from your function via attr(sys.function(), "<<attr.name>>"), and this trick is occasionally useful to avoid having to include multi-line text blocks in your function code; it's syntactically clearer, and avoids having to escape quotes, etc. mvbutils:::docskel shows one example.

fixr uses write.sourceable.function to create text files that use the flatdoc convention. Its counterpart FF reads these files back in after they're edited. The reading-in is not done with source but rather with source.mvb, which understands flatdoc. The call to doc=flatdoc causes the rest of the file to be read in as plain text, and assigned to the doc attribute of the function. Documentation can optionally be terminated before the end of the file with the following line:

```
<<end of doc>>
```

or whatever string is given as the argument to flatdoc; this line will cause source.mvb to revert to normal statement processing mode for the rest of the file. Note that vanilla source will not respect flatdoc; you do need to use source.mvb.

flatdoc should never be called from the command line; it should only appear in text files designed for source.mvb.

The rest of this section is probably obsolete, though things should still work.

If you are writing informal documentation for a group of functions together, you only need to flatdoc one of them, say myfun1. Informal help will work if you modify the others to e.g.

```
myfun2 <- structure( function(...) { whatever}, doc=list("myfun1"))
```

If you are writing with doc2Rd in mind and a number of such functions are to be grouped together, e.g. a group of "internal" functions in preparation for formal package release, you may find make.usage.section and make.arguments.section helpful.

Author(s)

Mark Bravington

See Also

source.mvb, doc2Rd, dochelp, write.sourceable.function, make.usage.section, make.arguments.section, fixr, the demo in "flatdoc.demo.R"

Examples

```
## Not run:
## Put next lines up to "<<end of doc>>" into a text file <<your filename>>
## and remove the initial hashes
#structure( function( x) {
#  x*x
#}
#,doc=flatdoc("<<end of doc>>"))
#
#Here is some informal documentation for the "SQUARE" function
#<<end of doc>>
## Now try SQUARE <- source.mvb( <<your filename>>); ?SQUARE
## Example with multiple attributes
## Put the next lines up to "<<end of part 2>>"
## into a text file, and remove the single hashes
#myfun <- structure( function( attname) {
```

```
#  attr( sys.function(), attname)
#}
#,  att1=flatdoc( EOF="<<end of part 1>>")
#,  att2=flatdoc( EOF="<<end of part 2>>"))
#This goes into "att1"
#<<end of part 1>>
#and this goes into "att2"
#<<end of part 2>>
## Now "source.mvb" that file, to create "myfun"; then:
myfun( 'att1') # "This goes into \\"att1\\""
myfun( 'att2') # "and this goes into \\"att2\\""

## End(Not run)
```

---

foodweb *Shows which functions call what*

Description

foodweb is applied to a group of functions (e.g. all those in a workspace); it produces a graphical display showing the hierarchy of which functions call which other ones. This is handy, for instance, when you have a great morass of functions in a workspace, and want to figure out which ones are meant to be called directly. callers.of(funs) and callees.of(funs) show which functions directly call, or are called directly by, funs.

Usage

```
foodweb( funs, where=1, charlim=80, prune=character(0), rprune,
    ancestors=TRUE, descendents=TRUE, plotting =TRUE, plotmath=FALSE,
    generics=c( "c","print","plot", "["), lwd=0.5, xblank=0.18,
    border="transparent", boxcolor="white", textcolor="black",
    color.lines=TRUE, highlight="red", ...)
## S3 method for class 'foodweb'
plot(x, textcolor, boxcolor, xblank, border, textargs = list(),
    use.centres = TRUE, color.lines = TRUE, poly.args = list(),
    expand.xbox = 1.05, expand.ybox = expand.xbox * 1.2, plotmath = FALSE,
    cex=par( "cex"), ...) # S3 method for foodweb
callers.of( funs, fw, recursive=FALSE)
callees.of( funs, fw, recursive=FALSE)
```

Arguments

funs	character vector OR (in foodweb only) the result of a previous foodweb call
where	position(s) on search path, or an environment, or a list of environments
charlim	controls maximum number of characters per horizontal line of plot
prune	character vector. If omitted, all funs will be shown; otherwise, only ancestors and descendants of functions in prune will be shown. Augments funs if required.

rprune	regexpr version of prune; prune <- funs %matching% rprune. Does NOT augment funs. Overrides prune if set.
ancestors	show ancestors of prune functions?
descendents	show descendents of prune functions?
plotting	graphical display?
plotmath	leave alone
generics	calls TO functions in generics won't be shown
lwd	see par
xblank	leave alone
border	border around name of each object (TRUE/FALSE)
boxcolor	background colour of each object's text box
textcolor	of each object
color.lines	will linking lines be coloured according to the level they originate at?
highlight	seemingly not used
cex	text size (see "cex" in ?par)
...	passed to plot.foodweb and thence to par
textargs	not currently used
use.centres	where to start/end linking lines. TRUE is more accurate but less tidy with big webs.
expand.xbox	how much horizontally bigger to make boxes relative to text?
expand.ybox	how much vertically bigger to ditto?
poly.args	other args to rect when boxes are drawn
fw	an object of class foodweb, or the funmat element thereof (see **Value**)
x	a foodweb (as an argument to plot.foodweb)
recursive	(callees.of and callers.of only) whether to include callee/rs of callee/rs of... (Thanks to William Proffitt for this suggestion.)

Details

The main value is in the graphical display. At the top ("level 0"), functions which don't call any others, and aren't called by any others, are shown without any linking lines. Functions which do call others, but aren't called themselves, appear on the next layer ("level 1"), with lines linking them to functions at other levels. Functions called only by level 1 functions appear next, at level 2, and so on. Functions which call each other will always appear on the same level, linked by a bent double arrow above them. The colour of a linking line shows what level of the hierarchy it came from.

foodweb makes some effort to arrange the functions on the display to keep the number of crossing lines low, but this is a hard problem! Judicious use of prune will help keep the display manageable. Perhaps counterintuitively, any functions NOT linked to those in prune (which all will be, by default) will be pruned from the display.

foodweb tries to catch names of functions that are stored as text, and it will pick up e.g. glm in do.call("glm", glm.args). There are limits to this, of course (?methods?).

The argument list may be somewhat daunting, but the only ones normally used are funs, where, and prune. Also, to get a readable display, you may need to reduce cex and/or charlim. A number of the less-obvious arguments are set by other functions which rely on plot.foodweb to do their display work. Several may disappear in future versions.

If the display from foodweb is unclear, try foodweb(.Last.value, cex=<<something below 1>>, charlim=<<something probably less than 100>>). This works because foodweb will also accept a foodweb-class object as its argument. You can also assign the result of foodweb to a variable, which is useful if you expect to do a lot of tinkering with the display, or to inspect the who-calls-whom matrix by hand.

callers.of and callees.of process the output of foodweb, looking for immediate dependencies only. The second argument will call foodweb by default, so it may be more efficient to call foodweb first and assign the result to a variable. NB you can set recursive=TRUE for the obvious result.

Bug in rgui windows graphics: When plotting the foodweb, there's a display bug in Rgui for windows which somehow causes the fontsize to shrink in each successive calls! Somehow par("ps") keeps on shrinking. Indeed, on my own machines, calling par(ps=par("ps"))$ps will show a decreasing value each time... Working around this was very tricky; variants of saving/restoring par *inside* plot.foodweb do not work. As of package **mvbutils** version 2.8.142, there's an attempted fix directly in foodweb, but conceivably the fixe will somehow cause problems for other people using default graphics windows in Rgui. Let me know if that's you... (in which case I'll add an option() to not apply the fix).

Value

foodweb returns an object of (S3) class foodweb. This has three components:

funmat	a matrix of 0s and 1s showing what (row) calls what (column). The dimnames are the function names.
x	shows the x-axis location of the centre of each function's name in the display, in par("usr") units
level	shows the y-axis location of the centre of each function's name in the display, in par("usr") units. For small numbers of functions, this will be an integer; for larger numbers, there will some adjustment around the nearest integer

Apart from graphical annotation, the main useful thing is funmat, which can be used to work out the "pecking order" and e.g. which functions directly call a given function. callers.of and callees.of return a character vector of function names.

Examples

```
foodweb( ) # functions in .GlobalEnv
# I have had to trim this set of examples because CRAN thinks it's too slow...
# ... though it's only 5sec on my humble laptop. So...
## Not run:
foodweb( where="package:mvbutils", cex=0.4, charlim=60) # yikes!
foodweb( c( find.funs("package:mvbutils"), "paste"))
# functions in .GlobalEnv, and "paste"
foodweb( find.funs("package:mvbutils"), prune="paste")
# only those parts of the tree connected to "paste";
# NB that funs <- unique( c( funs, prune)) inside "foodweb"
```

```
foodweb( where="package:mvbutils", rprune="aste")
# doesn't include "paste" as it's not in "mvbutils", and rprune doesn't augment funs
foodweb( where=asNamespace( "mvbutils")) # secret stuff
fw <- foodweb( where="package:mvbutils")

## End(Not run)
fw <- foodweb( where=asNamespace( "mvbutils")) # also plots
fw$funmat # a big matrix
callers.of( "mlocal", fw)
callees.of( "find.funs", fw)
# ie only descs of functions whose name contains 'name'
foodweb( where=asNamespace( 'mvbutils'), rprune="name", ancestors=FALSE, descendents=TRUE)
```

---

generic.dll.loader *Convenient automated loading of DLLs*

Description

generic.dll.loader is to be called from the .onLoad of a package. It calls library.dynam on all the DLLs it can find in the "libs" folder (so you don't need to specify their names), or in the appropriate sub-architecture folder below "libs". It also creates "R aliasses" in your namespace for all the *registered* low-level routines in each DLL (i.e. those returned by getDLLRegisteredRoutines, qv), so that the routines can be called efficiently later on from your code— see **Details**.

If you just want to use mvbutils to help build/maintain your package, and don't need your package to import/depend on other functions in mvbutils, then it's fine to just copy the code from generic.dll.loader etc and put it directly into your own .onLoad.

ldyn.tester, create.wrappers.for.dll, and ldyn.unload are to help you develop a DLL that has fully-registered routines, without immediately having to create an R package for it. ldyn.tester loads a DLL and returns its registration info. The DLL must be in a folder .../libs/<subarch> where <subarch> is .Platform$r_arch iff that is non-empty; this is because ldyn.tester merely tricks library.dynam into finding a spurious "package", and that's the folder structure that library.dynam needs to see. create.wrappers.for.dll does the alias-creation mentioned above for generic.dll.loader. ldyn.unload unloads the DLL.

Usage

```
# Only call this inside your .onLoad!
generic.dll.loader(libname, pkgname, ignore_error=FALSE)
# Only call these if you are informally developing a DLL outside a package
ldyn.tester(chname)
create.wrappers.for.dll( this.dll.info, ns=new.env( parent=parent.frame(2)))
ldyn.unload( l1)
```

Arguments

libname, pkgname
: as per .onLoad

ignore_error	?continue to load other DLLs if one fails?
chname	(for ldyn.tester) Path to the DLL (extension not required)
this.dll.info	(for create.wrappers.for.dll) A DLLInfo object, as returned by .dynLibs()[[N]] or library.dynam(...)
ns	(for create.wrappers.for.dll) If you're calling create.wrappers.for.dll manually, then this defaults to the calling environment, probably .GlobalEnv. For "internal use", ns is meant to be a namespace, but you shouldn't be using it like that!
l1	(for ldyn.unload) Result of previous call to ldyn.tester

Details

R-callable aliasses for your low-level routines will be called e.g. C_myrout1, Call_myrout2, F_myrout3, or Ext_myrout4, depending on type. Those for routines in "myfirstdll" will be stored in the environment LL_myfirstdll ("Low Level") in your package's namespace, which itself inherits from the namespace. In your own R code elsewhere in your package, you can then have something like

```
.C( LL_myfirstdll$C_myrout1, <<arguments>>) # NB no need for PACKAGE argument
```

Getting fancy, you can alternatively set the environment of your calling function to LL_myfirstdll (which inherits from the namespace, so all your other functions are still visible). In that case, you can just write

```
.C( C_myrout1, <<arguments>>)
```

Value

generic.dll.loader returns NULL (but see **Details**). ldyn.tester returns a class "DLLInfo" object if successful. ldyn.unload should return NULL if successful, and crash otherwise. create.wrappers.for.dll returns the environment containing the aliasses. Be careful with accidentally saving and loading the results of ldyn.tester and create.wrappers.for.dll; they won't be valid in a new R session. You might be better off creating them in the mvb.session.info environment on the search path; they will still be found, but won't persist in a different R session. See **Examples**.

See Also

set.finalizer for a safe way to ensure cleanup after low-level routines.

Examples

```
## Not run:
mypack:::.onLoad <- function( libname, pkgname) generic.dll.loader( libname, pkgname)
#... or just copy the code into your .onLoad
# For casual testing of a DLL that's not yet in a package
dl <- ldyn.tester( 'path/to/my/dll/libs/i386/mydll.dll')
getDLLRegisteredRoutines( l1)
LL_mydll <- create.wrappers.for.dll( dl)
```

```
.C( LL_mydll$C_rout1, as.integer( 0)) # ... whatever!
ldyn.unload( dl)
# Safer because not permanent:
assign( 'dl', ldyn.tester( 'path/to/my/dll/libs/i386/mydll.dll'), pos='mvb.session.info')
assign( 'LL_mydll', create.wrappers.for.dll( dl), pos='mvb.session.info')
.C( LL.mydll$C_rout1, as.integer( 0)) # ... whatever!

## End(Not run)
```

`get.backup` *Text backups of function source code*

Description

`get.backup` retrieves backups of a function or character object. `create.backups` creates backup files for all hitherto-unbacked-up functions in a search environment. For `get.backup` to work, all backups must have been created using the `fixr` system (or `create.backups`). `read.bkind` shows the names of objects with backups, and gives their associated filenames.

Usage

```
get.backup( name, where=1, rev=TRUE, zap.name=TRUE, unlength=TRUE)
create.backups( pos=1)
read.bkind( where=1)
```

Arguments

`name`	function name (character)
`where, pos`	position in search path (character or numeric), or e.g. `..mypack` for maintained package **mypack**.
`rev`	if TRUE, most recent backup comes first in the return value
`zap.name`	if TRUE, the tag `"funname" <-` at the start of each backup is removed
`unlength`	if TRUE, the first line of each backup is removed iff it consists only of a number equal to 1+length(object). This matches the (current) format of character object backups.

Details

`fixr` and `FF` are able to maintain text-file backups of source code, in a directory ".Backup.mvb" below the task directory. The directory will contain a file called "index", plus files BU1, BU2, etc. "index" shows the correspondence between function names and BUx files. Each BUx file contains multiple copies of the source code, with the oldest first. Even if a function is removed (or `move`d) from the workspace, its BUx file and "index" entry are not deleted.

The number of backups kept is controlled by `options(backup.fix)`, a numeric vector of length 2. The first element is how many backups to keep from the current R session. The second is how many previous R sessions to keep the final version of the source code from. Older versions get

discarded. I use `c(5,2)`. If you want to use the backup facility, you'll need to set this option in your `.First`. If the option is not set, no backups happen. If set, then every call to `Save` or `Save.pos` will create backups for all previously-unbackupped functions, by automatically calling `create.backups`. `create.backups` can also be called manually, to create the backup directory, index, and backup files for all functions in the currently-top task.

`get.backup` returns all available backup versions as **character vectors**, by default with the most recent first. To turn one of these character vectors into a function, a `source` step is needed; see **Examples**.

`read.bkind` shows which file to look for particular backups in. These files are text-format, so you can look at one in a text editor and manually extract the parts you want. You can also use `read.bkind` to set up a restoration-of-everything, as shown in **Examples**. I deliberately haven't included a function for mass restoration in `mvbutils`, because it's too dangerous and individual needs vary.

Currently there is no automatic way to determine the type of a backed-up object. All backups are stored as text, so text objects look very similar to functions. However, the first line of a text object is just a number equal to the length of the text object; the first line of a function object starts with "function(" or "structure(function(". The examples show one way to distinguish automatically.

The function `fix.order` uses the access dates of backup files to list your functions sorted by date order.

`move` will also move backup files and update INDEX files appropriately.

Value

`get.backup`	Either NULL with a warning, if no backups are found, or a list containing the backups, each as a character vector.
`create.backups`	NULL
`read.bkind`	a list with components `files` and `object.names`; these are character vector with elements in 1-1 correspondence. Some of the objects named may not currently exist in `where`.

Author(s)

Mark Bravington

See Also

`fixr`, `cd`, `move`

Examples

```
## Not run:
## Need some backups first
# Restore a function:
g1 <- get.backup( "myfun", "package:myfun")[[1]] # returns most recent backup only
# To turn this into an actual function (with source attribute as per your formatting):
myfun <- source.mvb( textConnection( g1)) # would be nice to have an self-closing t.c.
cat( get.backup( "myfun", "package:myfun", zap=FALSE)[[1]][1])
```

```
# shows "myfun" <- function...
# Restore a character vector:
mycharvec <- as.cat( get.backup( 'mycharvec', ..mypackage)[[1]]) # ready to roll
# Restore most recent backup of everything... brave!
# Will include functions & charvecs that have subsequently been deleted
bks <- read.bkind() # in current task
for( i in bks$object.names) {
  cat( "Restoring ", i, "...")
  gb <- get.backup( i, unlength=FALSE)[[1]] # unlength F so we can check type
  # Is it a charvec?
  if( grepl( '^ *[0-9]+ *$', gb[1])) # could check length too
    gb <- as.cat( gb[-1]) # remove line showing length and...
    # ...set class to "cat" for nice printing, as per 'as.cat'
  else {
    # Nope, so it's a function and needs to be sourced
    tc <- textConnection( gb)
    gbfun <- try( source.mvb( gb)) # will set source attribute, documentation etc.
    close( tc)
    if( gbfun %is.a% "try-error") {
      gbfun <- stop( function( ...) stop( ii %&% " failed to parse"), list( ii=i))
      attr( gbfun, 'source') <- gb # still assign source attribute
    }
    gb <- gbfun
  }
  assign( i, gb)
  cat( '\n')
}

## End(Not run)
```

---

hack *Modify standard R functions, including tweaking their default arguments*

Description

You probably shouldn't use these... hack lets you easily change the argument defaults of a function. `assign.to.base` replaces a function in `base` or `utils` (or any other package and its namespace and S3 methods table) with a modified version, possibly produced by hack. Package **mvbutils** uses these two to change the default position for library attachment, etc; see the code of `mvbutils:::.onLoad`.

Note that, if you call `assign.to.base` during the `.onLoad` of your package, then it must be called *directly* from the `.onLoad`, not via an intermediate function; otherwise, it won't correctly reset its argument in the import-environment of your namespace. To get round this, wrap it in an `mlocal`; see `mvbutils:::.onLoad` for an example.

`assign.to.base` is only meant for changing things in packages, e.g. not for things that merely sit in non-package environments high on the search path (where `<<-` should work). I don't know how it will behave if you try. It won't work for S4 methods, either.

Usage

```
hack( fun, ...)
assign.to.base( x, what=,  where=-1, in.imports=, override.env = TRUE)
```

Arguments

fun	a function (not a character string)
...	pairlist of arguments and new default values, e.g. arg1=1+2. Things on RHS of equal signs will **not** be evaluated.
x	function name (a character string)
what	function to replace x, defaulting to "replacement." %&% x
where	where to find the replacement function, defaulting to usual search path
in.imports	usually TRUE, if this is being called from an .onLoad method in a namespace. Make sure any copies of the function to be changed that are in the "imports" namespace also get changed. See **Description**.
override.env	should the replacement use its own environment, or (by default) the one that was originally there?

Examples

```
## Not run:
hack( dir, all.files=getOption( "ls.all.files", TRUE)) # from my '.First'
assign.to.base( "dir", hack( dir, all.files=TRUE))

## End(Not run)
```

---

help *The R help system*

Description

?x is the usual way to get help on x; it's primarily a shortcut for help(x). There are rarer but more flexible variations, such as x?y or help(x,...). See base-R help on help. The versions of help and ? exported by mvbutils behave exactly the same as base-R, unless base-R help *fails* after being called with a single argument, e.g. help(topic). In that case, if topic is an object with an attribute called "doc" (or failing that if topic or topic.doc is a character vector), then the attribute (or the character object) will be formatted and displayed by the pager (by default) or browser. This lets you write informal documentation for non-package objects that can still be found by help, and by colleagues you distribute your code to. See dochelp for more information. The rest of this documentation is copied directly from base-R for help, except as noted under **Arguments** for help_type.

Usage

```
help(topic, package = NULL, lib.loc = NULL,
     verbose = getOption("verbose"),
     try.all.packages = getOption("help.try.all.packages"),
     help_type = getOption("help_type"))
```

Arguments

topic	usually, a name or character string specifying the topic for which help is sought. A character string (enclosed in explicit single or double quotes) is always taken as naming a topic. If the value of topic is a length-one character vector the topic is taken to be the value of the only element. Otherwise topic must be a name or a reserved word (if syntactically valid) or character string. See **Details** for what happens if this is omitted.
package	a name or character vector giving the packages to look into for documentation, or NULL. By default, all packages in the search path are used. To avoid a name being deparsed use e.g. (pkg_ref) (see the examples).
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries. This is not used for HTML help (see **Details**).
verbose	logical; if TRUE, the file name is reported.
try.all.packages	logical; see **Note**.
help_type	character string:the type of help required. Possible values are "text", "html" and "pdf". Case is ignored, and partial matching is allowed. [Note that, for informal doco, getOption(mvb_help_type, "text") is used; i.e., the default there is always the pager, which lets you be as informal as you please.]

Details

The following types of help are available:

- Plain text help
- HTML help pages with hyperlinks to other topics, shown in a browser by browseURL. If for some reason HTML help is unavailable (see startDynamicHelp), plain text help will be used instead.
- For help only, typeset as PDF - see the section on **Offline help**.

The default for the type of help is selected when R is installed - the factory-fresh default is HTML help.

The rendering of text help will use directional quotes in suitable locales (UTF-8 and single-byte Windows locales): sometimes the fonts used do not support these quotes so this can be turned off by setting options(useFancyQuotes = FALSE).

topic is not optional. If it is omitted, R will give:

- If a package is specified, (text or, in interactive use only, HTML) information on the package, including hints/links to suitable help topics.
- If `lib.loc` only is specified, a (text) list of available packages.
- Help on `help` itself if none of the first three arguments is specified.

Some topics need to be quoted (by backticks) or given as a character string. These include those which cannot syntactically appear on their own such as unary and binary operators, `function` and control-flow reserved words (including `if`, `else` `for`, `in`, `repeat`, `while`, `break` and `next`). The other `reserved` words can be used as if they were names, for example `TRUE`, `NA` and `Inf`.

If multiple help files matching `topic` are found, in interactive use a menu is presented for the user to choose one: in batch use the first on the search path is used. (For HTML help the menu will be an HTML page, otherwise a graphical menu if possible if `getOption("menu.graphics")` is true, the default.)

Note that HTML help does not make use of `lib.loc`: it will always look first in the attached packages and then along `libPaths()`.

Offline help

Typeset documentation is produced by running the LaTeX version of the help page through `pdflatex`: this will produce a PDF file.

The appearance of the output can be customized through a file `Rhelp.cfg` somewhere in your LaTeX search path: this will be input as a LaTeX style file after `Rd.sty`. Some environment variables are consulted, notably `R_PAPERSIZE` (*via* `getOption("papersize")`) and `R_RD4PDF` (see `Making manuals` in the `R Installation and Administration Manual`).

If there is a function `offline_help_helper` in the workspace or further down the search path it is used to do the typesetting, otherwise the function of that name in the `utils` namespace (to which the first paragraph applies). It should accept at least two arguments, the name of the LaTeX file to be typeset and the type (which as from R 2.15.0 is ignored). As from R 2.14.0 it should accept a third argument, `texinputs`, which will give the graphics path when the help document contains figures, and will otherwise not be supplied.

Note

Unless `lib.loc` is specified explicitly, the loaded packages are searched before those in the specified libraries. This ensures that if a library is loaded from a library not in the known library trees, then the help from the loaded library is used. If `lib.loc` is specified explicitly, the loaded packages are *not* searched.

If this search fails and argument `try.all.packages` is `TRUE` and neither `packages` nor `lib.loc` is specified, then all the packages in the known library trees are searched for help on `topic` and a list of (any) packages where help may be found is displayed (with hyperlinks for `help_type = "html"`). **NB:** searching all packages can be slow, especially the first time (caching of files by the OS can expedite subsequent searches dramatically).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

? for shortcuts to help topics.

dochelp for how to write informal help with mvbutils.

help.search() or ?? for finding help pages on a vague topic; help.start() which opens the HTML version of the R help pages; library() for listing available packages and the help objects they contain; data() for listing available data sets; methods().

Use prompt() to get a prototype for writing help pages of your own package.

Examples

```
help()
help(help)              # the same
help(lapply)
help("for")             # or ?"for", but quotes/backticks are needed
help(package="splines") # get help even when package is not loaded
topi <- "women"
help(topi)
try(help("bs", try.all.packages=FALSE)) # reports not found (an error)
help("bs", try.all.packages=TRUE)       # reports can be found
                                        # in package 'splines'
## For programmatic use:
topic <- "family"; pkg_ref <- "stats"
help((topic), (pkg_ref))
```

---

help2flatdoc *Convert help files to flatdoc format.*

Description

Converts a vanilla R help file (as shown in the internal pager) to plain-text format. The output conventions are those in doc2Rd, so the output can be turned into Rd-format by running it through doc2Rd. This function is useful if you have existing Rd-format documentation and want to try out the flatdoc system of integrated code and documentation. Revised Nov 2017: now pretty good, but not perfect; see **Details**.

Usage

```
help2flatdoc( fun.name, pkg=NULL, text=NULL, aliases=NULL)
```

Arguments

fun.name	function name (a character string)
pkg	name of package
text	plain-text help
aliases	normally leave this empty— see **Details**. The real argument is text; if missing, this is deduced from the help for fun.name (need not be a function) in the installed package **pkg**.

Details

The package containing `fun.name` must be loaded first. If you write documentation using `flatdoc`, prepare the package with `pre.install`, build it with RCMD BUILD or INSTALL, and run `help2flatdoc` on the result, you should largely recover your original flat-format documentation. Some exceptions:

- Nesting in lists is ignored.
- Numbered lists won't convert back correctly (Nov 2017), but the problem there is in `doc2Rd`.
- Link-triggering phrases (i.e. that will be picked up by `doc2Rd`, such as "see <blah>") aren't explicitly created– probably, links could be automated better via an argument to `doc2Rd`.

Aliases (i.e. if this doco can be found by `help` under several different names) are deduced from function calls in the **Usage** section, in addition to anything supplied specifically in the `alias` argument. The latter is really just meant for internal use by `unpackage`.

See Also

`doc2Rd`

Examples

```
cd.doc <- help2flatdoc( "cd", "mvbutils")
print( cd.doc)
cd.Rd <- doc2Rd( cd.doc)
```

`install.pkg` *Package building, distributing, and checking*

Description

These are convenient wrappers for R's package creation and installation tools. They are designed to be used on packages created from tasks via `mvbutils` package, specifically `pre.install` (though they can be used for "home-made" packages). The `mvbutils` approach deliberately makes re-installation a rare event, and one call to `install.pkg` might suffice for the entire life of a simple package. After that very first installation, you'd probably only need to call `install.pkg` if (when...) new versions of R entail re-installation of packages, and `build.pkg/build.pkg.binary/check.pkg` when you want to give your package to others, either directly or via CRAN etc.

Folders: Source packages and built packages go into various folders, depending on various things. Normally you shouldn't have to mess around with the folder structure, but you will still need to *know* where built packages are put so that you can send them to other people. Specifically, these `...pkg...` functions work in the highest-versioned "Rx.y" folder that is not newer than the *running* R version. If no such folder exists, then 'build.pkg/build.pkg.binary" will create one from the running R version; you can also create such a folder manually, as a kind of "checkpoint", when you want to make your package dependent on a specific R version. See "Folders and R versions" in `mvbutils.packaging.tools` for full details.

There are also two minor housekeeping functions: `cull.old.builds` to tidy up detritus, and `set.rcmd.vars` which does absolutely nothing (yet). `cull.old.builds` looks through *all* "Rx.y" folders (where built packages live) and deletes the least-recent ".tar.gz" and ".zip" files in each (regardless of which built package versions are in the other "Rx.y" folders).

Usage

```
 # Usually: build.pkg( mypack) etc
install.pkg( pkg, character.only=FALSE, lib=.libPaths()[1], flags=character(0),
    multiarch=NA, preclean=TRUE)
build.pkg( pkg, character.only=FALSE, flags=character(0), cull.old.builds=TRUE)
 build.pkg.binary( pkg, character.only=FALSE, flags=character(0),
    cull.old.builds=TRUE, multiarch=NA, preclean=TRUE)
 check.pkg( pkg, character.only=FALSE, build.flags=character(0),
    check.flags=character( 0), CRAN=FALSE)
 cull.old.builds( pkg, character.only=FALSE)
 set.rcmd.vars( ...) # NYI; ...
 # ... if you need to set env vars eg PATH for R CMD to work,  then...
 # ... you have to do so yourself; see *Details*
```

Arguments

	See the examples
	usually an unquoted package name, but interpretation can be changed by non-default `character.only`. You can also get away with eg `..mypack`, ie a direct reference to the maintained package. A folder name can also be used, for a non-mvbutils-maintained package. Just as if it was "maintained", the folder should contain a subfolder with the (same) package name and the real package contents (eg "c:/r/mypack/mypack/DESCRIPTION" should exist), and any built things will go into eg "c:/r/mypack/R3.2"
`character.only`	
`pkg`	default FALSE. If TRUE, treat `pkg` as a normal object, which should therefore be a string containing the package's name. If `character.only` is itself a string, it will override `pkg` and be treated as the name of the package.
`lib`	(`install.pkg` only) where to install to; default is the same place R would install to, i.e. `.libPaths()[1]`.
`flags`	character vector, by default empty. Any entries should be function-specific flags, such as "–md5" for `build.pkg`. It will be passed through `paste( flags, collapse=" ")`, so you can supply flags individually (eg `flags=c( "--md5", "--compact.vignettes"))` or jointly (eg `flags="--md5 --compact.vignettes"`).
`build.flags, check.flags`	
	(`check.pkg` only) as per `flags` but for the two separate parts of `check.pkg` (see **Details**). `check.flags` is overridden if `CRAN==TRUE`'.
`preclean`	adds flag "–preclean" if TRUE (the default); this is probably a good idea since one build-failure can otherwise cause R to keep failing to build.
`multiarch`	Adds flag "-no-multiarch" if FALSE. Defaults to TRUE unless "Biarch:FALSE" is found in the DESCRIPTION. Default used to be FALSE when I was unable to get 64bit versions to build. Now I mostly can (after working round BINPREF64 bug in R3.3.something by futzing around in etc/arch/Makeconf based on random internet blogs).
`cull.old.builds`	
	self-explanatory

CRAN	(check.pkg only) if TRUE, set the --as-cran flag to "RCMD check" and unset all other check flags (except library locations, which are set automatically by all these functions). Note that this will cause R to check various internet databases, and so can be slow.
...	name-value pairs of system environment variables (not used for now)

Details

Before doing any of this, you need to have used pre.install to create a source package. (Or patch.install, if you've done all this before and just want to re-install/build/check for some reason.)

The only environment variable currently made known to R CMD is R_LIBS– let me know if others would be useful.

install.pkg calls "R CMD INSTALL" to install from a source package.

build.pkg calls "R CMD build" to wrap up the source package into a "tarball", as required by CRAN and also for distribution to non-Windows-and-Mac platforms.

build.pkg.binary (Windows & Mac only) calls "R CMD INSTALL –build" to generate a binary package. A temporary installation directory is used, so your existing installation is *not* overwritten or deleted if there's a problem; R CMD INSTALL –build has a nasty habit of doing just that unless you're careful, which build.pkg.binary is.

check.pkg calls "R CMD check" after first calling build.pkg (more efficiently, I should perhaps try to work out whether there's an up-to-date tarball already). It doesn't delete the tarball afterwards. It *may* also be possible for you to do some checks directly from R via functions in the **utils** package, which is potentially a lot quicker. However, NB the possibility of interference with your current R session. For example, at one stage codoc (which is the only check that I personally find very useful) tried to unload & load the package, which was very bad; but I think that may no longer be the case.

You *may* have to set some environment variables (eg PATH, and perhaps R_LIBS) for the underlying R CMD calls to work. Currently you have to do this manually— your .First or .Rprofile would be a good place. If you really object to changing these for the whole R session, let me know; I've left a placeholder for a function set.rcmd.vars that could store a list of environment variables to be set temporarily for the duration of the R CMD calls only, but I haven't implemented it (and won't unless there's demand).

Perhaps it would be desirable to let some flags be set automatically, eg via something in the pre.install.hook for a package. I'll add this if requested.

Value

Ideally, the "status code" of the corresponding RCMD operation: 0 for success or some other integer if not. It will have several attributes attached, most usefully "output" which duplicates what's printed while the functions are running. (Turn off "buffered output" in RGui to see it as it's happening.) This requires the existence of the "tee" shell redirection facility, which is built-in to Linux and presumably Macs, but not to Windows. You can get one version from Coreutils in GnuWin32; make sure this is on your PATH, but probably *after* the Rtools folders required by the R build process, to avoid conflicts between the other Coreutils versions and those in Rtools (I don't know what I'm talking about here, obviously; I'm just describing what I've done, which seems to work). If "tee" eventually moves to Rtools, then this won't be necessary. If no "tee" is available, then:

- progress of RCMD will be shown "live" in a separate shell window

- the status code is returned as NA, but still has the attributes including "output". You could, I suppose,

The point of all this "tee" business is that there's no reliable way in R itself to both show progress on-screen within R (which is useful, because these procedures can be slow) and to return the screen output as a character vector (which is useful so you can subsequently, pore through the error messages, or bask in a miasma of smugness).

Examples

```
## Not run:
# First time package installation
# Must be cd()ed to task above 'mvbutils'
maintain.packages( mvbutils)
pre.install( mvbutils)
install.pkg( mvbutils)
# Subsequent maintenance is all done by:
patch.install( mvbutils)
# For distro to
build.pkg( mvbutils)
# or on Windows (?and Macs?)
build.pkg.binary( mvbutils)
# If you enjoy R CMD CHECK:
check.pkg( mvbutils)
# Also legal:
build.pkg( ..mvbutils)
# To do it under programmatic control
for( ipack in all.my.package.names) {
  build.pkg( char=ipack)
}

## End(Not run)
```

---

library.dynam.reg *Auto-registration and loading of dynamic library*

Description

A bit like useDynLib but for direct use in your own package's .onLoad, this loads a DLL and creates objects that allow the DLL routines to be called directly. If your package "Splendid" calls library.dynam.reg in its .onLoad() to load a DLL "speedoo" which contains routines "whoosh" and "zoom", then an environment "C_speedoo" will be created in the asNamespace("Splendid"), and the environment will contain objects whoosh and zoom. R-code routines in "Splendid" can then call e.g.

```
.C( C_speedo$whoosh, ....)
```

You can only call `library.dynam.reg` inside `.onLoad`, because after that the namespace will be sealed so you can't poke more objects into it.

Note: Currently, *all* routines go into `C_speedoo`, regardless of how they are meant to be called (`.C`, `.Call`, `.Fortran`, or `.External`). It's up to you to call them the right way. I might change this to create separate `Call_speedoo` etc.

Note2: As of R3.1.1 at least, it's possible that "recent" changes to the `useDynLib` directive in a package namespace might obviate the need for this function. In particular, `useDynLib` can now create an environment/list that refers directly to DLL, containing references to individual routines (which will be slightly slowed because they need to be looked up each time). Also, `useDynLib` can automatically register its routines. What's not obvious is whether it can yet do both these things together— which is what `library.dynam.reg` is aimed at.

Usage

```
# Only inside a '.onLoad', where you will already know "package" and "lib.loc"
library.dynam.reg(chname, package, lib.loc, ...)
```

Arguments

`chname`	DLL name, a string— *without* any path
`package, lib.loc`	strings as for `library.dynam`
`...`	other args to `library.dynam`

`local.on.exit` *Macro-like functions*

Description

`local.on.exit` is the analogue of `on.exit` for "nested" or "macro" functions written with `mlocal`.

Usage

```
# Inside an 'mlocal' function of the form
# function( <<args>>, nlocal=sys.parent(), <<temp.params>>) mlocal({ <<code>> })
local.on.exit( expr, add=FALSE)
```

Arguments

`expr`	the expression to evaluate when the function ends
`add`	if TRUE, the expression will be appended to the existing `local.on.exit` expression. If FALSE, the latter is overwritten.

Details

on.exit doesn't work properly inside an mlocal function, because the scoping is wrong (though sometimes you get away with it). Use local.on.exit instead, in exactly the same way. I can't find any way to set the exit code in the **calling** function from within an mlocal function.

Exit code will be executed before any temporary variables are removed (see mlocal).

Author(s)

Mark Bravington

See Also

mlocal, local.return, local.on.exit, do.in.envir, and R-news 1/3

Examples

```
ffin <- function( nlocal=sys.parent(), x1234, yyy) mlocal({
  x1234 <- yyy <- 1 # x1234 & yyy are temporary variables
  # on.exit( cat( yyy)) # would crash after not finding yyy
  local.on.exit( cat( yyy))
  })
ffout <- function() {
  x1234 <- 99
  ffin()
  x1234 # still 99 because x1234 was temporary
}
ffout()
```

---

local.return *Macro-like functions*

Description

In an mlocal function, local.return should be used whenever return is called, wrapped inside the return call around the return arguments.

Usage

```
local.return(...) # Don't use it like this!
# Correct usage: return( local.return( ...))
```

Arguments

...	named and unnamed list, handled the same way as return before R 1.8, or as returnList

Author(s)

Mark Bravington

See Also

mlocal

Examples

```
ffin <- function( nlocal=sys.parent()) mlocal( return( local.return( a)))
ffout <- function( a) ffin()
ffout( 3) # 3
# whereas:
ffin <- function( nlocal=sys.parent()) mlocal( return( a))
ffout( 3) # NULL; "return" alone doesn't work
```

---

lsize *Report objects and their memory sizes*

Description

lsize is like ls, except it returns a numeric vector whose names are the object names, and whose elements are the object sizes. The vector is sorted in order of increasing size. lsize avoids loading objects cached using mlazy; instead of their true size, it uses the size of the file that stores each cached object, which is shown as a *negative* number. The file size is typically smaller than the size of the loaded object, because mlazy saves a compressed version. NB that lsize will scan *all* objects in the environment, including ones with funny names, whereas ls does so only if its all.names argument is set to TRUE.

Usage

```
lsize( envir=.GlobalEnv)
```

Arguments

envir	where to look for the objects. Will be coerced to environment, so that e.g. lsize(2) and lsize("package:mvbutils") work. envir can be a sys.frame– useful during debugging.

Value

Named numeric vector.

Author(s)

Mark Bravington

See Also

ls, mlazy

Examples

```
# Current workspace
lsize()
# Contrived example to show objects in a function's environment
{function(..., a, b, c) lsize( sys.frame( sys.nframe())) }()
# a, b, c are all missing; this example might break in future R versions
# ...   a   b   c
#  28  28  28  28
```

`maintain.packages` *Set up task package for live editing*

Description

See `mvbutils.packaging.tools` before reading or experimenting!

Set up task package(s) for editing and/or live-editing. Usually called in `.First` or `.First.task`. You need to be cded into the parent task of your task-package. `maintain.packages` must be called *before* loading the package via `library` or `require`. The converse, `unmaintain.package`, is rarely needed; it's really only meant for when `unpackage` doesn't work properly, and you want a "clean slate" task package.

Usage

```
# E.g. in your .First, after library( mvbutils), or in...
# ... a '.First.task' above yr task-package
maintain.packages(..., character.only = FALSE, autopatch=FALSE)
unmaintain.package( pkg, character.only = FALSE)
```

Arguments

`...`	names of packages, unquoted unless `character.only` is TRUE. Package names must correspond to subtasks of the current task.
`character.only`	see above
`pkg`	name of package, unquoted unless `character.only` is TRUE.
`autopatch`	whether to `patch.install` out-of-date installed packages (default FALSE, but TRUE is common).

Details

`maintain.packages( mypack)` loads a copy of your task-package "mypack" (as stored in its ".RData" file) into a environment `..mypack` (an "in-memory-task-package"), which itself lives in the "mvb.session.info" environment on the search path. You don't normally need to know this, because normally you'd modify/create/delete objects in the package via `fixr` or `fixr(..., pkg="mypack")` or `rm.pkg( ..., pkg="mypack")`. But to move objects between the package and other tasks, you do need to refer to the in-memory task package, e.g. via `move( ..., from=..Splendid, to=subtask/of/current)`.

In most cases, you will be prompted afterwards for whether to save the task package on disk, but you can always do yourself via `Save.pos( ..Splendid)`. Note that only these updates and saves only update the *task package* and the *loaded package*. To update the *source package* using the task package, call `pre.install`; to update the *installed package* on disk as well as the source package, call `patch.install`.

Creating new things: It's always safe to create new objects of any type in `.GlobalEnv`, then use `move(newthing,.,..mypack)`. For a new *function*, you can shortcut this two-step process and create it directly in the in-memory maintained package, via `fixr(..mypack$newfun)`; `fixr` will take care of synchronization with the loaded package. This also ought to work for text objects created via `fixtext`. Otherwise, use the two-step route, unless you have a good reason to do the following...

Directly modifying the maintained package: Rarely, you may have a really good reason to directly modify the contents of `..mypack`, e.g. via

```
..mypack$newfun <<- function( x) whatever
```

You can do it, but there are two problems to be aware of. The first is that changes won't be directly propagated to the loaded package, possibly not even after `patch.install` (though they will be honoured when you `library()` the package again). That is definitely the case for general data objects, and I'm not sure about functions; however, successful propagation after `patch.install` may happen for a special objects such as `mypack.DESCRIPTION` and documentation objects. Hence my general advice is to use `fixr` or `move`.

The second, minor, problem is that you will probably forget to use `<<-` and will use `<-` instead, so that a local copy of `..mypack` will be created in the current task. This is no big deal, and you can just `rm` the local copy; the local copy and the master copy in "mvb.session.info" both point to the same thing, and modifying one implies modifying the other, so that deleting the local copy won't lose your changes. `Save` detects accidental local copies of task packages, and omits them from the disk image, so there shouldn't be any problems next time you start R even if you completely forget about local/master copies.

Autopatch: If `autopatch==TRUE`, then `maintain.packages` will check whether the corresponding *installed* packages are older than the ".RData" files of the task packages. If they are, it will do a full `patch.install`; if not, it will still call `patch.install` but only to reverse-update any bundled DLLs (see `pre.install`), not to re-install the R-source. I find `autopatch` useful with packages containing C code, where a crash in the C code can cause R to die before the most recent R-code changes have been "committed" with `patch.install`. When you next start R, a call to `maintain.packages` with `autopatch=TRUE` will "commit" the changes *before* the package is loaded, because you have to call `maintain.packages` before `library`; this seems to be more reliable than running `patch.install` manually after `library` after a restart.

Maintained packages as tasks

If you use `mvbutils` to pre-build your package, then your package must exist as a task in the `cd` hierarchy. Older versions of `mvbutils` allowed you to `cd` to a maintained package, but this is now forbidden because of the scope for confusion. Thanks to `maintain.packages`, there is no compelling need to have the package/task at the top of the search path; `fixr`, `move`, etc work just fine without. If you really do want to `cd` to a maintained package, you must call `unmaintain.package` first.

One piece of cleanup that I recommend, is to move any subtasks of "mypack" one level up in the task hierarchy, and to remove the tasks object from "Splendid" itself, e.g. via something like:

```
cd( task.above.splendid)
tasks <- c( tasks, combined.file.paths( tasks[ "Splendid"], ..Splendid$tasks))
# ... combined.file.paths is an imaginary function. Watch out if you've used relative paths!
rm.pkg( tasks, pkg="Splendid")
```

See Also

mvbutils.packaging.tools, fixr, pre.install, patch.installed, unpackage

Examples

```
## Not run:
# In your .First:
library( mvbutils)
maintain.packages( myfirstpack, mysecondpack, mythirdpack)
# or...
live.edit.list <- c( 'myfirstpack', 'mysecondpack', 'mythirdpack')
maintain.packages( live.edit.list, character.only=TRUE)
library( myfirstpack) # etc

## End(Not run)
```

---

make.NAMESPACE *Auto-create a NAMESPACE file*

Description

Called by pre.install for would-be packages that have a .onLoad function, and are therefore assumed to want a namespace. Produces defaults for the import, export, and S3Methods. You can modify this information prior to the NAMESPACE file being created, using the pre-install hook mechanism. The default for import is taken from the DESCRIPTION file, but the defaults for export and S3 methods are deduced from your functions, and are described below.

Usage

```
# Don't call this directly-- pre.install will do it automatically for you
make.NAMESPACE( env=1, path=attr( env, "path"),
 description=read.dcf( file.path( path, "DESCRIPTION"))[1,], more.exports=character( 0))
```

Arguments

env	character or numeric position on search path
path	directory where proto-package lives
description	(character) elements for the DESCRIPTION file, e.g. c(..., Author="R.A. Fisher", ...). By default, read from existing file.
more.exports	(character) things to export that normally wouldn't be.

Details

There is (currently) no attempt to handle S4 methods.

The imported packages are those listed in the "Depends:" and "Imports:" field of the DESCRIPTION file. At present, all functions in those packages will be imported (i.e. no "importFrom" provision).

The exported functions are all those in `find.documented(doctype="any")` unless they appear to be S3 methods, plus any functions that have a non-NULL `export.me` attribute. The latter is a cheap way of arranging for a function to be exported, but without formal documentation (is that wise??). `pre.install` will incorporate any undocumented `export.me` functions in the "mypack-internal.Rd" file, so that RCMD CHECK will be happy.

The S3 methods are all the functions whose names start "«generic».." and whose first argument has the same name as in the appropriate `<<generic>>`. The generics that are checked are (i) the names of the character vector `.knownS3Generics` in package **base**; (ii) all functions that look like generics in any importees or dependees of your would-be package (i.e. functions in the namespace whose name is a prefix of a function in the S3 methods table of the namespace, and whose body contains a call to `UseMethod`); (iii) any plausible-looking generic in your would-be package (effectively the same criterion). Documented functions which look like methods but whose flat-doc documentation names them explicitly in the **Usage** section (e.g. referring to `print.myclass(...)` rather than just `print(...)`, the latter being how you're supposed to document methods) are assumed not be methods.

See Also

`pre.install`, `flatdoc`

---

make_dull *Hide dull columns in data frames*

Description

`make_dull` AKA `make.dull` adds a "dull" S3 class to designated columns in a `data.frame`. When the `data.frame` is printed, entries in those columns will show up just as "...". Useful for hiding long boring stuff like nucleotide sequences, MD5 sums, and filenames. Columns will still print clearly if manually extracted.

The `dull` class has methods for `format` (used when printing a `data.frame`) and `[`, so that dullness is perpetuated.

Usage

```
make_dull(df, cols)
```

Arguments

`df`	a data.frame
`cols`	columns to designate

Details

Ask yourself: do you *really* want details of a function called `make_dull`? Life may be sweet but it is also short.

More details: `make_dull` is both autologous and idempotent.

Value

A modified data.frame

Examples

```
# Becos more logical syntax:
rsample <- function (n = length(pop), pop, replace = FALSE, prob = NULL){
  pop[sample(seq_along(pop) - 1, size = n, replace = replace, prob = prob) + 1]
}
df <- data.frame( x=1:3,
    y=apply( matrix( rsample( 150, as.raw( 33:127), rep=TRUE), 50, 3), 2, rawToChar),
    stringsAsFactors=FALSE) # s.A.F. value shouldn't matter
df # zzzzzzzzzzzzzz
df <- make_dull( df, 'y')
df # wow, exciting!
df$y # zzzzzzzzzzzzz
```

`max_pkg_ver` *Max package version*

Description

Finds the highest version number of an installed package in (possibly) *several* libraries. Mainly for internal use in `mvbutils`, but might come in handy if your version numbers have gotten out-of-synch eg with different R versions. On my setup, all my "non-base" libraries are folders inside "d:/rpackages", with folder names such as "R2.13"; my `.First` sets `.libPaths()` to all of these that are below the running version of R (but that are still legal for that R version; so for R > 3.0, folders named "R2.xxxx" would be excluded). Hence I can call `max_pkg_ver( mypack, "d:/rpackages")` to find the highest installed version in all these subfolders.

Usage

```
max_pkg_ver(pkg, libroot, pattern = "^[rR][ -]?[0-9]+")
    # NB named with underscores to avoid interpretation as S3 method
```

Arguments

`pkg`	character, the name of the package
`libroot`	folder(s) to be searched recursively for package **pkg**
`pattern`	what regexp to use when looking for potential libraries to recurse into

Value

A numeric_version object for the highest-numbered installation, with value numeric_version("0") if no such package is found. If libroot is a single library containing the package, the result will equal packageVersion(pkg, limbroot).

Examples

```
max_pkg_ver( "mvbutils", .libPaths())
```

---

mcut *Put reals and integers into specified bins, returning factors.*

Description

Put reals and integers into specified bins, returning ordered factors. Like cut but for human use.

Usage

```
mcut( x, breaks, pre.lab='', mid.lab='', post.lab='', digits=getOption( 'digits'))
mintcut( x, breaks=NULL, prefix='', all.levels=, by.breaks=1)
```

Arguments

x	(numeric vector) What to bin– will be coerced to integer for mintcut
breaks	(numeric vector) LH end of each bin– should be increasing. Values of x exactly on the LH end of a bin will go into that bin, not the previous one. For mintcut, defaults to equal-size bins across the range of x, where bin size is set from by.breaks which itself defaults to 1. For mcut, should start with -Inf if necessary, but should not finish with Inf unless you want a bin for Infs only.
prefix, pre.lab	(string) What to prepend to the factor labels– e.g. "Amps" if your original data is about Amps.
mid.lab	"units" to append to numeric vals *inside* factor labels. Tends to make the labels harder to read; try using post.lab instead.
post.lab	(string) What to append to the factor labels.
digits	(integer) How many digits to put into the factor labels.
all.levels	if FALSE, omit factor levels that don't occur in x. To override "automatically", just set the "all.levels" attribute of breaks to anything non-NULL; useful e.g. if you are repeatedly calling mintcut with the same breaks and you always want all.levels=TRUE.
by.breaks	for mintcut when default breaks is used, to set the bin size.

Details

Values of x below breaks[1] will end up as NAs. For mintcut, factor labels (well, the bit after the prefix) will be of the form "2-7" or "3" (if the bin range is 1) or "8+" (for last in range). For mcut, labels will look like this (apart from the pre.lab and post.lab bits): "[<0.25]" or "[0.25,0.50]" or "[>=0.75]".

Examples

```
set.seed( 1)
mcut( runif( 5), c( 0.25, 0.5, 0.75))
# [1] [0.25,0.50] [0.25,0.50] [0.50,0.75] [>=0.75]     [<0.25]
# Levels: [<0.25] [0.25,0.50] [0.50,0.75] [>=0.75]
 mcut( runif( 5), c( 0.25, 0.5, 0.75), pre.lab='A', post.lab='B', digits=1)
# [1] A[>=0.8]B    A[>=0.8]B    A[0.5,0.8]B A[0.5,0.8]B A[<0.2]B
# Levels: A[<0.2]B A[0.2,0.5]B A[0.5,0.8]B A[>=0.8]B
mintcut( 1:8, c( 2, 4, 7))
# [1] <NA> 2-3  2-3  4-6  4-6  4-6  7+   7+
# Levels: 2-3 4-6 7+
mintcut( c( 1, 2, 4)) # auto bins, size defaulting to 1
# [1] 1  2  4+
# Levels: 1 < 2 < 3 < 4+
mintcut( c( 1, 2, 6), by=2) # auto bins of size 2
# [1] 1-2 1-2 5+
# Levels: 1-2 < 3-4 < 5+
```

---

mlazy *Cacheing objects for lazy-load access*

Description

mlazy and friends are designed for handling collections of biggish objects, where only a few of the objects are accessed during any period, and especially where the individual objects might change and the collection might grow or shrink. As with "lazy loading" of packages, and the gdata/ASOR packages, the idea is to avoid the time & memory overhead associated with loading in numerous huge R binary objects when not all will be needed. Unlike lazy loading and gdata, mlazy caches each mlazyed object in a separate file, so it also avoids the overhead that would be associated with changing/adding/deleting objects if all objects lived in the same big file. When a workspace is Saved, the code updates only those individual object files that need updating.

mlazy does not require any special structure for object collections; in particular, the data doesn't have to go into a package. mlazy is particularly useful for users of cd because each cd to/from a task causes a read/write of the binary image file (usually ".RData"), which can be very large if mlazy is not used. Read DETAILS next. Feedback is welcome.

Usage

```
mlazy( ..., what, envir=.GlobalEnv, save.now=TRUE)
  # cache some objects
mtidy( ..., what, envir=.GlobalEnv)
```

```
  # (cache and) purge the cache to disk, freeing memory
demlazy( ..., what, envir=.GlobalEnv)
  # makes 'what' into normal uncached objects
mcachees( envir=.GlobalEnv)
  # shows which objects in  envir are cached
attach.mlazy( dir, pos=2, name=)
  # load mcached workspace into new search environment,
  # or create empty s.e. for cacheing
```

Arguments

...	unquoted object names, overridden by what if supplied
what	character vector of object names, all from the same environment. For mtidy and demlazy, defaults to all currently-cached objects in envir
envir	environment or position on the search path, defaulting to the environment where what or objs live.
save.now	see DETAILS
dir	name of directory, relative to task.home.
pos	numeric position of environment on search path, 2 or more
name	name to give environment, defaulting to something like "data:current.task:dir".

Value

These functions are used only for their side-effects, except for cachees which returns a character vector of object names.

More details

All this is geared to working with saved images (i.e. ".RData" or "all.rda" files) rather than creating all objects anew each session via source. If you use the latter approach, mlazy will probably be of little value.

The easiest way to set up cacheing is just to create your objects as normal, then call

mlazy(<<objname1>>, <<objname2>>, <<etc>>)

Save()

This will not seem to do much immediately– your object can be read and changed as normal, and is still taking up memory. The memory and time savings will come in your next R session in this workspace.

You should never see any differences (except in time & memory usage) between working with cached (AKA mlazyed) and normal uncached objects.[One minor exception is that cacheing a function may stuff up the automatic backup system, or at any rate the "backstop" version of it which runs when you cd. This is deliberate, for speeding up cd. But why would you cache a *function* anyway?]

mlazy itself doesn't save the workspace image (the ".RData" or "all.rda" file), which is where the references live; that's why you need to call Save periodically. save.image and save will **not** work properly, and nor will load– see NOTE below. Save doesn't store cached objects directly in the

".RData" file, but instead stores the uncached objects as normal in `.RData` together with a special object called something like `.mcache00` (guaranteed not to conflict with one of your own objects). When the `.RData` file is subsequently reloaded by `cd`, the presence of the `.mcache00` object triggers the creation of "stub" objects that will load the real cached objects from disk when and only when each one is required; the `.mcache00` object is then deleted. Cached objects are loaded & stored in a subdirectory "mlazy" from individual files called "obj*.rda", where "*" is a number.

`mlazy` and `Save` do not immediately free any memory, to avoid any unnecessary re-loading from disk if you access the objects again during the current session. To force a "memory purge" *during* an R session, you need to call `mtidy`. `mtidy` purges its arguments from the cache, replacing them by `promises` just as when loading the workspace; when a reference is next accessed, its cached version will be re-loaded from disk. `mtidy` can be useful if you are looping over objects, and want to keep memory growth limited– you can `mtidy` each object as the last statement in the loop. By default, `mtidy` purges the cache of all objects that have previously been cached. `mtidy` also caches any formerly uncached arguments, so one call to `mtidy` can be used instead of `mlazy( ...); mtidy( ...)`.

`move` understands cached objects, and will shuffle the files accordingly.

`demlazy` will **delete** the corresponding "obj*.rda" file(s), so that only an in-memory copy will then exist; don't forget to `Save` soon after.

Warning: The system function `load` does not understand cacheing. If you merely `load` an image file saved using `Save`, cached objects will not be there, but there will be an extra object called something like `.mcache00`. Hence, if you have cached objects in your ROOT task, they will not be visible when you start R until you load the `mvbutils` library– another fine reason to do that in your `.First`. The `.First.lib` function in `mvbutils` calls `setup.mcache( .GlobalEnv)` to automatically prepare any references in the ROOT task.

Cacheing in other search environments: It is possible to cache in search environments other the current top one (AKA the current workspace, AKA `.GlobalEnv`). This could be useful if, for example, you have a large number of simulated datasets that you might need to access, but you don't want them cluttering up `.GlobalEnv`. If you weren't worried about cacheing, you'd probably do this by calling `attach( "<<filename>>")`. The cacheing equivalent is `attach.mlazy( "cachedir")`. The argument is the name of a directory where the cached objects will be (or already are) stored; the directory will be created if necessary. If there is a ".RData" file in the directory, `attach.mlazy` will load it and set up any references properly; the ".RData" file will presumably contain mostly references to cached data objects, but can contain normal uncached objects too.

Once you have set up a cacheable search environment via `attach.mlazy` (typically in search position 2), you can cache objects into it using `mlazy` with the `envir` argument set (typically to 2). If the objects are originally somewhere else, they will be transferred to `envir` before cacheing. Whenever you want to save the cached objects, call `Save.pos(2)`.

You will probably also want to modify or create the `.First.task` (see `cd`) of the current task so that it calls `attach.mlazy("<<cache directory name>>")`. Also, you should create a `.Last.task` (see `cd`) containing `detach(2)`, otherwise `cd(..)` and `cd(0/...)` won't work.

Options: By default, `mlazy` now saves & loads into a auto-created subdirectory called "mlazy". In the earliest releases, though, it saved "obj*.rda" files into the same directory as ".RData". It will now **move** any "obj*.rda" files that it finds alongside ".RData" into the "mlazy" subdirectory.

You can (possibly) override this by setting `options( mlazy.subdir=FALSE)`, but the default is likely more reliable.

By default, there is no way to figure out what object is contained in a "obj*.rda" without forcibly loading that file or inspecting the `.mcache00` object in the "parent" `.RData` file– not that you should ever need to know. However, if you set `options( mlazy.index=TRUE)` (**recommended**), then a file "obj.ind" will be maintained in the "mlazy" directory, showing (object name - value) pairs in plain text (tab-separated). For directories with very large numbers of objects, there may be some speed penalty. If you want to create an index file for an existing "mlazy" directory that lacks one, `cd` to the task and call `mvbutils:::mupdate.mcache.index.if.opt(mlazy.index=TRUE)`.

See `Save` for how to set compression options, and `save` for what you can set them to; `options(mvbutils.compression_le` may save some time, at the expense of disk space.

Troubleshooting: In the unlikely event of needing to manually load a cached image file, use `load.refdb`– `cd` and `attach.mlazy` do this automatically.

In the unlikely event of lost/corrupted data, you can manually reload individual "obj*.rda" files using `load`– each "obj*.rda" file contains one object stored with its correct name. Before doing that, call `demlazy( what=mcachees())` to avoid subsequent trouble. Once you have reloaded the objects, you can call `mlazy` again.

See **Options** for the easy way to check what object is stored in a particular "obj*.rda" file. If that feature is turned off on your system, the failsafe way is to load the file into a new environment, e.g. `e <- new.env(); load( "obj99.rda", e); ls( e)`.

To see how memory changes when you call `mlazy` and `mtidy`, call `gc()`.

To check object sizes *without* actually loading the cached objects, use `lsize`. Many functions that iterate over all objects in the environment, such as `eapply`, will cause `mlazy` objects to be loaded.

Housekeeping of "obj**.rda" files happens during `Save`; any obsolete files (i.e. corresponding to objects that have been `removed`) are deleted.

Inner workings: What happens: each workspace acquires a `mcache` attribute, which is a named numeric vector. The absolute values of the entries correspond to files– 53 corresponds to a file "obj53.rda", etc., and the names to objects. When an object `myobj` is `mlazyed`, the `mcache` is augmented by a new element named "myobj" with a new file number, and that file is saved to disk. Also, "myobj" is replaced with an active binding (see `makeActiveBinding`). The active binding is a function which retrieves or sets the object's data within the function's environment. If the function is called in change-value mode, then it also makes negative the file number in `mcache`. Hence it's possible to tell whether an object has been changed since last being saved.

When an object is first `mlazyed`, the object data is placed directly into the active binding function's environment so that the function can find/modify the data. When an object is `mtidyed`, or when a cached image is loaded from disk, the thing placed into the A.B.fun's environment is not the data itself, but instead a `promise` saying, in effect, "fetch me from disk when you need me". The promise gets forced when the object is accessed for reading or writing. This is how "lazy loading" of packages works, and also the **gdata** package. However, for `mlazy` there is the additional requirement of being able to determine whether an object has been modified; for efficiency, only modified objects should be written to disk when there is a `Save`.

There is presumably some speed penalty from using a cache, but experience to date suggests that the penalty is small. Cached objects are saved in compressed format, which seems to take a little longer than an uncompressed save, but loading seems pretty quick compared to uncompressed files.

Author(s)

Mark Bravington

See Also

lsize, gc, package **gdata**, package **ASOR**

Examples

```
## Not run:
biggo <- matrix( runif( 1e6), 1000, 1000)
gc() # lots of memory
mlazy( biggo)
gc() # still lots of memory
mtidy( biggo)
gc() # better
biggo[1,1]
gc() # worse; it's been reloaded

## End(Not run)
```

---

mlocal *Macro-like functions*

Description

mlocal lets you write a function whose statements are executed in its caller's frame, rather than in its own frame.

Usage

```
# Use only as wrapper of function body, like this:
# my.fun <- function(..., nlocal=sys.parent()) mlocal( expr)
# ... should be replaced by the arguments of "my.fun"
# expr should be replaced by the code of "my.fun"
# nlocal should always be included as shown
mlocal( expr) # Don't use it like this!
```

Arguments

expr	the function code, normally a braced expression

Details

Sometimes it's useful to write a "child" function that can create and modify variables in its parent directly, without using `assign` or `<<-` (note that `<<-` will only work on variables that exist already). This can make for clearer, more modular programming; for example, tedious initializations of many variables can be hidden inside an `initialize()` statement. The definition of an `mlocal` function does not have to occur within its caller; the `mlocal` function can exist as a completely separate R object.

`mlocal` functions can have arguments just like normal functions. These arguments will temporarily hide any objects of the same name in the `nlocal` frame (i.e. the calling frame). When the `mlocal` function exits, its arguments will be deleted from the calling frame and the hidden objects (if any) will be restored. Sometimes it's desirable to avoid cluttering the calling frame with variables that only matter to the `mlocal` function. A useful convention is to "declare" such temporary variables in your function definition, as defaultless arguments after the `nlocal` argument.

The `nlocal` argument of an `mlocal` function– which must ALWAYS be included in the definition, with the default specified as `sys.parent()`– can normally be omitted when invoking your `mlocal` function. However, you will need to set it explicitly when your function is to be called by another, e.g. `lapply`; see the third example. A more daring usage is to call e.g. `fun.mlocal(nlocal=another.frame.number)` so that the statements in `fun.mlocal` get executed in a completely different frame. A convoluted example can be found in the (internal) function `find.debug.HQ` in the **debug** package, which creates a frame and then defines a large number of variables in it by calling `setup.debug.admin(nlocal=new.frame.number)`. As of 2016, you can also set `nlocal` to be an environment.

`mlocal` functions can be nested, though this gets confusing. By default, all evaluation will happen in the same frame, that of the original caller.

Note that (at least at present) all arguments are evaluated as soon as your `mlocal` function is invoked, rather than by the usual lazy evaluation mechanism. Missing arguments are still OK, though.

If you call `return` in an `mlocal` function, you must call `local.return` too.

`on.exit` doesn't work properly. If you want to have exit code in the `mlocal` function itself, use `local.on.exit`. I can't find any way to set the exit code in the calling function from within an `mlocal` function. (Not checked for some years)

Frame-dependent functions (sys.parent()) etc. will not do what you expect inside an `mlocal` function. For R versions between at least 1.8 and 2.15, calling the `mvb...` versions will return information about the **caller** of the current `mlocal()` function caller (or the original caller, if there is a chain of `mlocals`). For example, `mvb.sys.function()` returns the definition of the caller, and `mvb.sys.parent()` the frame of the caller's parent. Note that `sys.frame( mvb.sys.nframe())` gives the current environment (i.e. where all the variables live), because this is shared between the caller and the `mlocal` function. Other behaviour seems to depend on the version of R, and in R 2.15 I don't know how to access the definition of the `mlocal` function itself. This means, for example, that you can't reliably access attributes of the `mlocal` function itself, though you can access those of its caller via e.g. `attr( mvb.sys.function(), "thing")`.

Value

As per your function; also see `local.return`.

Author(s)

Mark Bravington

See Also

local.return, local.on.exit, do.in.envir, localfuncs, and R-news 1/3 2001 for a related approach to "macros"

Examples

```
# Tidiness and variable creation
init <- function( nlocal=sys.parent()) mlocal( sqr.a <- a*a)
ffout <- function( a) { init(); sqr.a }
ffout( 5) # 25
# Parameters and temporary variables
ffin <- function( n, nlocal=sys.parent(), a, i) mlocal({
    # this "n" and "a" will temporarily replace caller's "n" and "a"
    print( n)
    a <- 1
    for( i in 1:n)
      a <- a*x
    a
  })
x.to.the.n.plus.1 <- function( x, n) {
    print( ffin( n+1))
    print( n)
    print( ls())
  }
x.to.the.n.plus.1( 3, 2) # prints as follows:
# [1] 3 (in "ffin")
# [1] 27 (result of "ffin")
# [1] 2 (original n)
# [1] "n" "x" (vars in "x.to.the..."-- NB no a or i)
# Use of "nlocal"
ffin <- function( i, nlocal=sys.parent()) mlocal( a <- a+i )
ffout <- function( ivec) { a <- 0; sapply( ivec, ffin, nlocal=sys.nframe()) }
ffout( 1:3) # 1 3 6
```

---

move *Organizing R workspaces*

Description

move shifts one or more objects around the task hierarchy (see cd), whether or not the source and destination are currently attached on the search path.

Usage

```
# Usually: unquoted object name, unquoted from and to, e.g.
# move( thing, ., 0/somewhere)
# Use 'what' arg to move several objects at once, e.g.
# move( what=c( "thing1", "thing2"), <<etc>>)
# move( x, from, to)
```

```
# move( what=, from, to)
# Next line shows the formal args, but the real usage would NEVER be like this...0
move( x='.', from='.', to='.', what, overwrite.by.default=FALSE, copy=FALSE)
```

Arguments

x	unquoted name
from	unquoted path specifier (or maintained package specifier)
to	unquoted path specifier (or M.P. specifier)
what	character vector
overwrite.by.default	logical(1)
copy	logical(1)

Details

The normal invocation is something like `move( myobj, ., 0/another.task)`– note the lack of quotes around `myobj`. To move objects with names that have to be quoted, or to move several objects at the same time, specify the `what` argument: e.g. `move( what=c( "myobj", "%myop%"), ., 0/another.task)`. Note that `move` is playing fast and loose with standard argument matching here; it correctly interprets the `.` as `from`, rather than `x`. This well-meaning subversion can lead to unexpected trouble if you deviate from the paradigms in **Examples**. If in doubt, you can always name `from` and `to`.

`move` can also handle moves in and out of packages being live-edited (see `maintain.packages`). If you want to specify a move to/from your package "whizzbang", the syntax of `to` and `from` should be `..whizzbang` (i.e. the actual environment where the pre-installed package lives). An alternative for those short of typing practice is `maintained.packages$whizzbang`. No quotes in either case.

If `move` finds an object with the same name in the destination, you will be asked whether to overwrite it. If you say no, the object will not be moved. If you want to force overwriting of a large number of objects, set `overwrite.by.default=TRUE`.

By default, `move` will delete the original object after it has safely arrived in its destination. It's normally only necessary (and more helpful) to have just one instance of an object; after all, if it needs to be accessed by several different tasks, you can just `move` it to an ancestral task. However, if you really do want a duplicate, you can avoid deletion of the original by setting `copy=TRUE`.

You will be prompted for whether to `save` the source and destination tasks, if they are attached somewhere, but not in position 1. Normally this is a good idea, but you can always say no, and call `Save.pos` later. If the source and/or destination are not attached, they will of course be saved automatically. The top workspace (i.e. current task) `.GlobalEnv` is never saved automatically; you have to call `Save` yourself.

`move` is not meant to be called within other functions.

Author(s)

Mark Bravington

See Also

cd

Examples

```
## Not run:
move( myobj, ., 0) # back to the ROOT task
move( what="%myop%", 0/first.task, 0/second.task)
# neither source nor destination attached. Funny name requires "what"
move( what=c( "first.obj", "second.obj"), ., ../sibling.task)
# multiple objects require "what"
move( myobj, ..myfirstpack, ..mysecondpack) # live-edited packages

## End(Not run)
```

`multirep` *Replacement and insertion functions with more/less than 1 replacement per spot*

Description

`multirep` is like `replace`, but the replacements are a list of the same length as the number of elements to replace. Each element of the list can have 0, 1, or more elements– the original vector will be expanded/contracted accordingly. (If all elements of the list have length 1, the result will be the same length as the original.) `multinsert` is similar, but doesn't overwrite the elements in `orig` (so the result of `multinsert` is longer). `massrep` is like `multirep`, but takes lists as arguments so that a group-of-line-numbers in the first list is replaced by a group-of-lines in the second list.

Usage

```
multirep( orig, at, repl, sorted.at=TRUE)
multinsert( orig, at, ins, sorted.at=TRUE)
massrep( orig, atlist, replist, sorted.at=TRUE)
```

Arguments

`orig`	vector
`at`	numeric vector, saying which elements of the original will be replaced or appended-to. Can't exceed `length(orig)`. 0 is legal in `multinsert` but not `multirep`. Assumed sorted unless `sorted.at` is set to FALSE.
`atlist`	list where each element is a group of line numbers to be replaced by the corresponding element of `replist` (and that element can have a different length). Normally each group of line numbers would be consecutive, but this is not mandatory.

`repl, ins, replist`
: a list of replacements. `repl[[i]]` will replace line `at[i]` in `orig`, possibly removing it (if `repl[[i]]` has length 0) or inserting extra elements (if `repl[[i]]` has length > 1). In `multinsert`, `repl` can be a non-list, whereupon it will be cast to `list(repl)` [if `at` is length 1] or `as.list(repl)` [if `at` is length>1]. If `length(repl)` < `length(at)`, `repl` will be replicated to the appropriate size. If `repl` is atomic, it will be typecast into a list– in this case, all replacements/insertions will be of length 1.

`sorted.at`
: if TRUE, then `at` had better be sorted beforehand; if FALSE, `at` will be sorted for you inside `multirep`, and `repl` is reordered accordingly.

Examples

```
multirep( cq( the, cat, sat, on, the, mat), c( 2, 6),
    list( cq( big, bug), cq( elephant, howdah, cushion)))
# [1] "the" "big" "bug" "sat" "on" "the" "elephant" "howdah" "cushion"
multirep( cq( the, cat, sat, on, the, mat), c( 2, 6),
    list( cq( big, bug), character(0)))
# [1] "the" "big" "bug" "sat" "on" "the"
# NB the 0 in next example:
multinsert( cq( cat, sat, on, mat), c( 0, 4),
    list( cq( fat), cq( cleaning, equipment)))
# [1] "fat" "cat" "sat" "on" "mat" "cleaning" "equipment"
```

---

mvb.sys.parent *Functions to Access the Function Call Stack*

Description

These functions are "do what I mean, not what I say" equivalents of the corresponding system functions. The system functions can behave strangely when called in strange ways (primarily inside `eval` calls). The `mvb` equivalents behave in a more predictable fashion.

Usage

```
mvb.sys.parent(n=1)
mvb.sys.nframe()
mvb.parent.frame(n=1)
mvb.eval.parent( expr, n=1)
mvb.match.call(definition = sys.function(mvb.sys.parent()),
   call = sys.call(mvb.sys.parent()), expand.dots = TRUE, envir= mvb.parent.frame( 2))
mvb.nargs()
mvb.sys.call(which = 0)
mvb.sys.function(n)
```

Arguments

	All as per the corresponding system functions, from whole helpfiles the following is taken:
	the frame number if non-negative, the number of generations to go back if negative. (See the Details section.)
which	the number of frame generations to go back.
definition	a function, by default the function from which match.call is called.
call	an unevaluated call to the function specified by definition, as generated by call.
expr	an expression to evaluate
expand.dots	logical. Should arguments matching ... in the call be included or left as a ... argument?
envir	an environment from which the ... in call are retrieved, if any (as per base::match.call)

Details

Sometimes eval is used to execute statements in another frame. If such statements include calls to the system versions of these routines, the results will probably not be what you want. In technical terms: the same environment will actually appear several times on the call stack (returned by sys.frame()) but with a different calling history each time. The mvb. equivalents look through sys.frames() for the first frame whose environment is identical to the environment they were called from, and base all conclusions on that first frame. To see how in detail, look at the most fundamental function: mvb.sys.parent.

mvbutils pre 2.7 used to include mvb.sys.on.exit as well (to return whatever the on.exit code would be), but I think this was by mistake; the code was actually specific to my debug package (which already has its own substitute), and so I've moved it out of mvbutils.

Value

See the helpfiles for the system functions.

Author(s)

Mark Bravington

See Also

sys.parent, sys.nframe, parent.frame, eval.parent, match.call, nargs, sys.call, sys.function

Examples

```
ff.no.eval <- function() sys.nframe()
ff.no.eval() # 1
ff.system <- function() eval( quote( sys.nframe()), envir=sys.frame( sys.nframe()))
ff.system() # expect 1 as per ff.no.eval, get 3
ff.mvb <- function() eval( quote( mvb.sys.nframe()), envir=sys.frame( sys.nframe()))
ff.mvb() # 1
```

```
ff.no.eval <- function(...) sys.call()
ff.no.eval( 27, b=4) # ff.no.eval( 27, b=4)
ff.system <- function(...) eval( quote( sys.call()), envir=sys.frame( sys.nframe()))
ff.system( 27, b=4) # eval( expr, envir, enclos) !!!
ff.mvb <- function(...) eval( quote( mvb.sys.call()), envir=sys.frame( sys.nframe()))
ff.mvb( 27, b=4) # ff.mvb( 27, b=4)
```

`mvbutils.operators` *Utility operators*

Description

Succinct or convenience operators

Usage

```
a %&% b
x %**% y
a %!in% b
vector %except% condition
x %grepling% patt
x %is.not.a% what
x %is.a% what
x %is.not.an% what
x %is.an% what
x %matching% patt
a %not.in% b
a %not.in.range% b
x %perling% patt
x %that.match% patt
x %that.dont.match% patt
a %that.are.in% b
x %without.name% what
a %in.range% b
a %such.that% b
a %SUCH.THAT% b
from %upto% to
from %downto% to
x %where% cond
x %where.warn% cond
a %<-% value # really e.g. {x;y} %<-% list( 'yes', sqrt(pi)) to create x & y
```

Arguments

`a`, `b`, `vector`, `condition`, `x`, `y`, `what`, `patt`, `from`, `to`, `cond`, `value`
see **Arguments by function**.

Value

`%&%`	character vector. If either is zero-length, so is the result (unlike `paste`).
`%**%`	numeric, possibly a matrix
`%upto%, %downto%`	numeric
`%is.a%, %in%, etc`	logical
`%<-%`	technically NULL return, but it overwrites / creates objects; see below...
`All others`	same type as first argument.

Arguments by function

`%&%` a, b: character vectors to be `pasted` with no separator. If either is zero-length, so is the result (unlike `paste`).

`%**%` x, y: matrices or vectors to be multiplied using `%*%` but with less fuss about dimensions

`%!in%`, `%that.are.in%` a, b: vectors (character, numeric, complex, or logical).

`%except%` vector, condition: character or numeric vectors

`%in.range%`, `%not.in.range%` a, b: numeric vectors.

`%is.a%`, etc. x: object whose class is to be checked

`%is.a%`, etc. what: class name

`%matching%`, `%that.match%`, `%that.dont.match%`, `%grepling%`, `%perling%` x: character vector

`%matching%`, `%that.match%`, `%that.dont.match%`, `%grepling%`, `%perling%` patt: character vector of regexps, with perl syntax for `%perling%`

`%such.that%`, `%SUCH.THAT%` a: vector

`%such.that%`, `%SUCH.THAT%` b: expression containing a `.`, to subscript `a` with

`%upto%`, `%downto%` from, to: numeric(1)

`%where%`, `%where.warn%` x: data.frame

`%where%`, `%where.warn%` cond: unquoted expression to be `eval`ed in context of `x`, then in the calling frame of `%where%` (or `.GlobalEnv`). Should evaluate to logical (or maybe numeric or character); NA is treated as FALSE. Wrap `cond` in parentheses to avoid trouble with operator precedence.

`%without.name%` x: object with `names` attribute

`%without.name%` what: character vector of names to drop

`%<-%` a, value: `value` should be a list, and `a` should be e.g. `{x;y;z}` with as many elements as `value` has. The elements of `value` are assigned, in order, to the objects named in `a`, which are created / overwritten in the calling environment.

Author(s)

Mark Bravington

See Also

bquote

Examples

```
"a" %&% "b" # "ab"
matrix( 1:4, 2, 2) %**% matrix( 1:2, 2, 1) # c( 7, 10); '%*%' gives matrix result
matrix( 1:2, 2, 1) %**% matrix( 1:4, 2, 2) # c( 5, 11); '%*%' gives error
1:2 %**% matrix( 1:4, 2, 2) # '%*%' gives matrix result
1:5 %!in% 3:4 # c( TRUE, TRUE, FALSE, FALSE, TRUE)
1:5 %not.in% 3:4 # c( TRUE, TRUE, FALSE, FALSE, TRUE)
1:5 %that.are.in% 3:4 # c( 3, 4)
trf <- try( 1+"nonsense")
if( trf %is.not.a% "try-error") cat( "OK\n") else cat( "not OK\n")
1:5 %except% c(2,4,6) # c(1,3,5)
c( alpha=1, beta=2) %without.name% "alpha" # c( beta=2)
1:5 %in.range% c( 2, 4) # c(F,T,T,T,F)
1:5 %not.in.range% c( 2, 4) # c(T,F,F,F,T)
c( "cat", "hat", "dog", "brick") %matching% c( "at", "ic") # cat hat brick
c( "cat", "hat", "dog", "brick") %that.match% c( "at", "ic") # cat hat brick; ...
# ... synonym for '%matching%'
c( "cat", "hat", "dog", "brick") %that.dont.match% c( "at", "ic") # dog; ...
# ... like '%except%' but for regexps
1 %upto% 2 # 1:2
1 %upto% 0 # numeric( 0); use %upto% rather than : in for-loops to avoid unintended errors
1 %downto% 0 # 1:0
1 %downto% 2 # numeric( 0)
ff <- function( which.row) {
    x <- data.frame( a=1:3, b=4:6)
    x %where% (a==which.row)
  }
ff( 2) # data.frame( a=2, b=5)
x <- data.frame( start=1:3, end=c( 4, 5, 0))
x %where.warn% (start < end) # gives warning about row 3
(1:5) %such.that% (.>2) # 3,4,5
listio <- list( a=1,  b=2)
chars <- cq( a, b)
chars %SUCH.THAT% (listio[[.]]==2) # 'b'; %such.that% won't work because [[]] can't handle xtuples
{x;y} %<-% list( 'yes', sqrt(pi))
# x: [1] "yes"
# y: [1] 1.772
```

`mvbutils.packaging.tools`

How to create & maintain packages with mvbutils

Description

This document covers:

- using `mvbutils` to create a new package from scratch;
- using `mvbutils` to maintain a package you've created (e.g. edit it while using it);
- converting an existing package into `mvbutils`-compatible format;

- how to customize the package-creation process.

For clarity, the simplest usage is presented first in each case. For how to do things differently, first look further down this document, then in the documentation for `pre.install` and perhaps `doc2Rd`.

You need to understand `cd` and `fixr` before trying any of this.

Setting up a package from scratch

First, the simplest case: suppose you have some pure R code and maybe data that you'd like to make into a package called "Splendid". The bare-minimum steps you need are:-

- Make sure all the code & data lives in a single task called "Splendid".
- `cd` to the task above "Splendid"
- `maintain.packages( Splendid)`
- `pre.install( Splendid)`. This will create a "source package" in a subdirectory of Splendid's task directory. The subdirectory will be called "Splendid".
- Make sure you have all the R build tools installed and on your path– see "R-exts" for details (and NB that if you need to install Latex, then google MikTex & choose a *minimal* install).
- `install.pkg( Splendid)` to do what you'd expect. On Windows, you can alternatively first do `build.pkg.binary( Splendid)`, then use R's menus to "Packages/Install from local zip files".
- `library(Splendid)`; your package will be loaded for use, and is also ready for live-editing.

Your package will probably just about work now, but the result won't yet be perfect. The additional steps you'll likely need are these:

- Sort out the **Description file or object**[(]see below[)]
- Provide **Documentation and metadata**[(]see below[)]
- Sort out any C/Fortran source code, pre-compiled code, demos, and other additional files (see `pre.install`)
- Move any subtasks of Splendid to one level up the task hierarchy (see `maintain.packages`)

Once you have set up "Splendid" so that `maintain.packages` works, you won't need to `cd` directly into "Splendid" again— which is good, because you're not allowed to.

Glossary: *Task package* is a folder with at least an ".RData" file, linked into the `cd` hierarchy. It contains master copies of the objects in your package, plus perhaps a few other objects required to build the package (e.g. stand-alone items of documentation).

In-memory task package is an environment in the current R session that contains an image of the task package. Objects in it are never used directly, only as templates for editing. It is loaded by `maintain.packages`, and `Save.pos` uses it to update the task package (usually automatic).

Source package is a folder containing, yes, an R-style source package. It is created initially by `pre.install`, and subsequently by `patch.install` or `pre.install`.

Installed package is a folder containing, yes, an R-style installed package. It is always created from the source package, initially by `install.pkg` and subsequently by `patch.install` or `install.pkg`.

Loaded package is the in-memory version of an installed package, loaded by `library`.

Tarball package is a zipped-up version of a source package, for distro on non-Windows-Mac platforms or submission to CRAN and subsequent installation via "R CMD INSTALL". Usually it will not contain DLLs of any low-level code, just the source low-level code. It is created by `build.pkg`.

Binary package is a special zipped-up version for distro to Windows or Macs that includes actual DLLs, for installation via e.g. the "Packages/Install from local ZIP" menu. It is created by `build.pkg.binary`.

Built package is a tarball package or binary package.

Converting an existing package

Suppose you have already have a *source* package "hardway", and would like to try maintaining it via `mvbutils`. You'll need to create a task package, then create a new version of the source package, then re-install it. The first step is to call `unpackage( hardway)` to creat the task package "hardway" in a subdirectory of the current task. Plain-text documentation will be attached to functions, or stored as ".doc" text objects. All functions and documentation must thereafter be edited using `fixr`. The full sequence is something like:

```
# Create task package in subdirectory of current:
unpackage( "path/to/existing/source/package/hardway")
#
# Load image into memory:
maintain.packages( hardway)
#
# Make new version of source package:
pre.install( hardway, ...) # use dir= to control where new source pkg goes
#
install.pkg( hardway) # or build.pkg.binary( hardway) followed by "install from local zip file" menu
#
library( hardway) # off yer go
```

If you get problems after `maintain.packages`, you might need `unmaintain.package( hardway)` to clear out the in-memory copy of the new task package.

Documentation and metadata

Documentation for functions can be stored as plain text just after a function's source code, as described in `flatdoc`. Just about anything will do– you don't absolutely have to follow the conventional structure of R help if you are really in a hurry. However, the easiest way to add kosheR but skeletal documentation to your function `brilliant`, is `fixr( brilliant, new.doc=TRUE)`; again, see `flatdoc` and `doc2Rd` if you want to understand what's going on. The format is almost exactly as displayed in plain-text help, i.e. from `help(..., help_type="text")`. My recommendation is to just start writing something that looks reasonable, and see if it works. To quickly test the ultimate appearance, you can use e.g. `docotest(..Splendid$brilliant)`. More generally, run `patch.install(Splendid)` which, as explained in **Maintaining a package** below, updates everything for your package including the help system, so you can then just do `?brilliant`. If you run into problems with writing documentation for your functions, then refer to `doc2Rd` for further details of format, such as how to document several functions in the same file.

You can also provide three other types of documentation, for: (i) general use of your package (please do! it helps the user a lot; packages where the doco PDF consists only of an alphabetical list of functions/objects are a pain); (ii) more specific aspects of usage that are not tied to individual functions, such as this file; and (iii) datasets. These types of documentation should be stored in the package as text objects whose name ends in ".doc"; examples of the three types could be "Splendid.package.doc", "glitzograms.with.Splendid.doc", and "earlobes.doc" if you have a dataset `earlobes`. See `doc2Rd` for format details.

You must document every function and dataset that the user will see, but you don't need to document any others. The foregoing applies iff your package has a **Namespace**, which it must for R 2.14 up.

Description file or object: When you first create a package from a task via `pre.install`, there probably won't be any DESCRIPTION information, so `mvbutils` will create a default "DESCRIPTION" file in your task folder, which it then copies to the source package. However, the default won't really be what you really want, as you'll realize if you type `library( help=Splendid)`. You can either manually edit the default "DESCRIPTION" file, or you can use `fixtext(Splendid.DESCRIPTION, pkg="Splendid")` to create a text object in your task package, which you then populate with the contents of the default "DESCRIPTION" file, and then edit. If a `Splendid.DESCRIPTION` object exists, `mvbutils` will use it in preference to a file; I find this tidier, because more of the package metadata lives in a single place, viz. inside the task package.

Apart from the obvious changes needed to the default "DESCRIPTION" file or text object, the most important fields to add are "Imports:" (or "Depends:" for packages that are pre-R2.14 and that also don't have a namespace), to say what other packages are needed by "Splendid". The DESCRIPTION file/text should rarely need to be updated, since the "autoversion" feature (see `pre.install` doco) can be used to take care of version numbering. The most common reason to change the DESCRIPTION is probably to add/remove packages in "Imports"; at present, this pretty much requires you to unload & reload the package, but I may try to expedite this in future versions.

Vignettes: In time, I plan to get `mvbutils` working nicely with `knitr`. At present (Jan 2013), the easiest way to create vignettes with `mvbutils` is to produce your own "homebrewed" PDFs however you prefer, and put them into the "inst/doc" folder. `pre.install/patch.install` will sort them out and link them into the help system. To provide more information than the filename, use `fixtext` to create a text object in your task package called e.g. `mypack.VIGNETTES`, with lines as follows:

```
my.first.vignette: Behold leviathan, mate
my.second.vignette: What a good idea, to write a vignette
```

As a *very* experimental feature, you can also include R code for a homebrewed vignette, via a file with the same name but extension ".R" also in "inst/doc". Users can access it as normal for vignette code, via `edit( vignette( "my.first.vignette", package="mypack"))` or via doing something to `system.file( file.path( "doc", "my.first.vignette.R"), package="mypack")`. You can put full-on Sweave-style vignettes into a "vignettes" folder, and they should be set up correctly in the source package. Currently, though, they are **not** re-installed by `patch.install`; you need to use `build.pkg` and `install.pkg` (partly defeating the point of these package-building utilities).

Very technical details about homebrewed vignettes: "Rnw stubs" are created for all homebrewed vignettes so that the help system finds them. A rudimentary index will be created for vignettes not mentioned in `<<mypack>>.VIGNETTES`. If you create your own "inst/doc/index.html" file, this takes precedence over mvbutil's versions, so that `<<mypack>>.VIGNETTES` is not used.

Namespace: Usually this is automatic. pre.install etc automatically creates a "NAMESPACE" file for your package, ensuring inter alia that all documented objects are user-visible. To load DLLs, add a .onLoad function that contains the body code of generic.dll.loader in package **mvbutils** (thus avoiding dependence on mvbutils). For more complicated fiddling, see **Customizing package creation**.

Packages without namespaces pre r 2 14: Namespaces only became compulsory with R 2.14. If you're setting up your package in an earlier version of R, mvbutils will *not* create a namespace unless it finds a .onLoad function. To trigger namespacing, just create a .onLoad with this definition: function(libname, pkgname) {}.

Maintaining a package

Once you have successfully gotten your "Splendid" package installed and loaded the first time, you should rarely need to call install.pkg or build.pkg etc again, except when you are about to distribute to others. In your own work, after calling maintain.packages and library in an R session, you can modify, add and delete functions, datasets, and documentation in your package via the standard functions fixr, move, and rm.pkg (or directly), and these changes will mostly be immediately manifested in the loaded package within your R session– this is "live editing". The changes are made first to the in-memory task package, which will be called e.g. ..Splendid, and then propagated to the loaded package. Don't try to manipulate the loaded package's namespace directly. See maintain.packages for details.

To update the installed package (on disk), call patch.install(Splendid); this also calls pre.install to update the source package, updates the help system in the current session, and does a few other synchronizations. You need to call patch.install before quitting R to ensure that the changes are manifest in the loaded package the next time you start R; otherwise they will only exist in the in-memory task package, and won't be callable.

Troubleshooting: In rare cases, you may find that maintain.packages(Splendid) fails. If that happens, there won't be a ..Splendid environment, which means you can't fix whatever caused the load failure. The load failure is (invariably in my experience) caused by a hidden attempt to load a namespaced package, which is failing for yet another reason, usually something in its .onLoad; that package might or might not be "Splendid" itself. If you can work out what other package is trying to load itself– say badpack– you can temporarily get round the problem by making use of the character vector partial.namespaces, which lives in the "mvb.session.info" search environment, as follows:

```
partial.namespaces <<- c( partial.namespaces, "badpack")
```

That will prevent execution of badpack:::.onLoad. Consequently badpack won't be properly loaded, but at least the task package will be loaded into ..Splendid, so that you can make a start on the problem. If you can't work out which package is causing the trouble, try

```
partial.namespaces <<- "EVERY PACKAGE"
```

After that, no namespaced package will load properly, so remember to clear partial.namespaces <<- NULL before resuming normal service.

Occasionally (usually during patch.install), you might see R errors like "cannot allocate vector of size 4.8Gb". I think this happens when some internal cache gets out-of-synch. It doesn't seem to cause much damage to the installed package, but once it's happened in an R session, it tends to happen again. I usually quit & restart R.

You might also find find.lurking.envs useful, via eapply(..Splendid, find.lurking.envs); this will show any functions (or other things) in ..Splendid that have accidentally acquired a non-standard environment such as a namespace, which can trigger a "hidden" package load attempt. The environment for all functions in ..Splendid *should* probably be .GlobalEnv; the environments in the *loaded* package will be different, of course.

It's rare to need to manually inspect either the source package or the installed package. But if you do, then spkg helps for the former, e.g. dir(spkg(mypack)); and system.file helps for the latter, e.g. system.file(package="mypack"), or system.file(file.path("help", "AnIndex"), package="mypack").

Distributing and checking

build.pkg calls R CMD BUILD to create a "tarball" of the package (a ".tar.gz" file), which is the appropriate format for distribution to Unix folk and submission to CRAN. build.pkg.binary creates a binary package (a ".zip" file), suitable for Windows or Macs. check.pkg runs R CMD CHECK (but see next paragraph for a quicker alternative), which is required by CRAN and sometimes useful at other times. These .pkg functions are pretty simple wrappers to the R CMD tools with similar names. However, for those with imperfect memories and limited time, there are enough arcane and mutable nuances with the "raw" R CMD commands (including the risk of inadvertently deleting existing installations) to make the wrappers in mvbutils useful.

Various functions in the **tools** package can be used to quickly check specific aspects of an *installed* package, without needing a full-on, and slow, R CMD CHECK. In particular, I sometimes use

```
codoc( spkg( mypack)) # also spkg( "mypack"), spkg( ..mypack)
undoc( spkg( ..mypack))
```

Nothing is printed unless a problem is found, so a blank result is good news! It's also possible to run other tools such as checkTnF and checkFF similarly.

By default, mvbutils adds code to the source package to circumvent the CRAN checks for "no visible function/binding", which I consider to be a waste of time; for example, unless circumvented they generate 338 false positives for package **mvbutils**. If for some reason you actually want these checks, see "Overriding defaults" in pre.install.

Folders and different r versions

Life can get complicated when there are several versions of R around, particularly when they require different package formats at source or build or install time (eg R 2.10, 2.12, R 3.0). install.pkg etc do their best to simplify this for you. You won't normally need to know the details unless you are trying to maintain several versions of your package for different versions of R for distribution to other people who use those different R versions. But if you do need to know the details, then the default folder structure is as follows. If the task package lives in folder "mypack", then the source package is created by pre.install in "mypack/mypack", and the built package(s) will go into folders such as "mypack/R2.15" depending on what R version is running.

Note that your *task package* can only ever have one version; if different behaviour is required for different R versions, then you need to code this up your functions, or via some trickery in .onLoad.

Built packages: Building comes first: the tarballed/zipped packages from build.pkg and build.pkg.binary are placed in a folder parallel to the source package, with a name of the

form "Rx.y". mvbutils tries to be sensible about what "x.y" should be. It will never be newer than the *running* R version. It will never be older than the most recent major R version that required mandatory package rebuilds (eg R 3.0 and R 2.12). If one or more folders already exist that satisfy those properties, the highest-numbered one will be used. If not, a new folder will be created with the current R major version (eg R 2.15.3 will trigger a folder "R2.15"). You can create your own "Rx.y" folder, for instance if the current version of your package requires an R feature only found in R version "x.y". Also, mvbutils knows which R versions change the format of built packages, and will create a new folder for such a version if required.

The default behaviour is therefore that build.pkg.<binary> will keep building into the same folder. For example, if at some point a "mypack/R2.12" folder was created, then that's where all builds will be sent regardless of the running R version, until you either manually create an "mypack/Rx.y" folder that's closer to the running R version, or the latter hits 3.0 which automatically triggers the creation of a new "mypack/R3.0" folder. Thanks to the "autoversion" feature of pre.install, the version number of the build will change whenever <pre/patch>.install is used. (Note that old built packages are not removed until/unless you explicitly call cull.old.builds, although it's "good housekeeping" to do the latter occasionally.) By manually creating new "Rx.y" folder when necessary, you can ensure that there *won't* be any updates to built packages for R older than "x.y", which gives a kind of "checkpoint" feature; your built packages for older versions of R (ie for distribution to users of those older R versions) won't be accidentally zapped by cull.old.builds housekeeping, and you can be sure that old code running under old versions of R will still work.

What this does *not* let you do easily, is use your current R version to create updated versions of your package for R-versions that pre-date the most up-to-date "Rx.y" folder. For example, if you are running R3.0, there is guaranteed to be an "R3.0" folder, so calling build.pkg<.binary> won't build new packages in an "R2.15" folder. Again, usually this doesn't matter, because new "Rx.y" folders are only rarely created automatically, so builds will tend to stay in the same folder and the newest version will be accessible to all. But sometimes it is a hassle... Nevertheless, I have managed to maintain parallel versions of my packages across the R2.15-R3.0 change, by (sequentially) running two R versions and calling build.pkg<.binary> from each. (Note that build.pkg<.binary> can only build in the format of running R version– you can't "cross-build" for different built formats from the same R session.)

Source packages: R occasionally demands a change in *source* package format, as opposed to *built* package format (as with R 3.0). (IIRC one example is R 2.10, with the change in helpfile format.) Then you face the problem of how to keep several source packages. This can be controlled by options("mvbutils.sourcepkgdir.postfix"), which is appended to the name of the folder where your source package will be created and used for building or installing. The default is the empty string "", so that the default source package folder for "mypack" is "mypack/mypack". To allow for multiple source package versions, you could put something like this in your .First or ".Rprofile":

```
if( getRversion() >= numeric_version( '4.0')) {
  # New source package format
  options( mvbutils.sourcepkgdir.postfix='[R4]')
}
```

Everything *should* then work automatically; all source-package operations will refer to "mypack/mypack[R4]" if you are running version 4 or above, or to "mypack/mypack" if you are running an earlier R version, and you should never really need to know the source package foldername yourself (build.pkg etc do it all for you). This depends on you setting the option yourself,

and *has not been tested* yet. Eventually I may hardwire the feature automatically into `mvbutils` (or is it better for each source package to go into an appropriate built-package folder? but that sounds a bit like version hell).

Customizing package creation

You can customize many aspects of the **mvbutils** package-creation process, by adding a function `pre.install.hook.Splendid` to your package. See `pre.install` for further details.

`mvbutils.utils` *Miscellaneous utilities*

Description

Miscellaneous utilities.

Usage

```
as.cat( x)
clip( x, n=1)
cq( ...)
deparse.names.parsably( x)
empty.data.frame( ...)
env.name.string( env)
expanded.call( nlocal=sys.parent())
everyth( x, by=1, from=1)
find.funs(pos=1, ..., exclude.mcache = TRUE, mode="function")
find.lurking.envs(obj, delve=FALSE, trace=FALSE)
index( lvector)
integ(expr, lo, hi, what = "x", ..., args.to.integrate = list())
is.dir( dir)
isF( x)
isT( x)
legal.filename( name)
lsall( ...)
masked( pos)
masking( pos=1)
mkdir( dirlist)
most.recent( lvec)
my.all.equal( x, y, ...)
named( x)
nscat( fmt, ..., sep='\n', file='')
nscatn( fmt, ..., sep='\n', file='')
option.or.default( opt.name, default=NULL)
pos( substrs, mainstrs, any.case = FALSE, names.for.output)
put.in.session( ...)
returnList( ...)
```

```
safe.rbind( df1, df2) # Deprecated in 2013
scatn( fmt, ..., sep='\n', file='')
to.regexpr( x)
yes.no( prompt, default)
```

Arguments

x, y, n, ..., by, env, from, exclude.mcache, nlocal, lvector, dir, name, pos, frame, mode, dirlist, lvec, o
: see "Arguments by function"

Details

as.cat makes a character vector print as if it was catted rather than printed (one element per line, no extra quotes or backslashes, no [1] etc prefixes).

clip removes the last n elements of x.

cq is handy for typing cq(alpha, beta, gamma) instead of cq("alpha", "beta", "gamma"). Certain strings DO still require quotes around them, e.g. cq("NULL", "1-2")).

deparse.names.parsably is like deparse except that name objects get wrapped in a call to as.name, so that they won't be evaluated accidentally.

empty.data.frame creates a template data frame with 0 rows but with all columns of the appropriate type. Useful for rbinding to later.

env.name.string returns a string naming an environment; its name attribute if there is one, or the name of its path attribute if applicable, concatenated with the first line of what would be shown if you printed the argument. Unlike environmentName, this will always return a non-empty string.

expanded.call returns the full argument list available to its caller, including defaults where arguments were not set explicitly. The arguments may not be those originally passed, if they were modified before the invocation of expanded.call. Default arguments which depend on calculations after the invocation of expanded.call will lead to an error.

everyth extracts every by-th element of x, starting at position from.

find.funs finds "function" objects (or objects of other modes, via the "mode" arg) in one or more environments, optionally matching a pattern.

find.lurking.envs(myobj) will search through myobj and all its attributes, returning the size of each sub-object. The size of environments is returned as Inf. The search is completely recursive, except for environments and by default the inner workings of functions; attributes of the entire function are always recursed. Changing the delve parameter to TRUE ensures full recursion of function arguments and function bodies, which will show e.g. the srcref structure; try it to see why the default is FALSE. find.lurking.envs can be very useful for working out e.g. why the result of a model-fitting function is taking up 1000000MB of disk space; sometimes this is due to unnecessary environments in well-concealed places.

index returns the position(s) of TRUE elements. Unlike which: attributes are lost; NA elements map to NAs; index(<<length 0 object>>) is numeric(0); index(<<non-logical>>) is NA.

integ is a handy wrapper for integrate, that takes an expression rather than a function— so integ(sin(x), 0, 1) "just works".

is.dir tests for directoriness.

isF and isT test a logical *scalar* in the obvious way, with NA (and non-logicals) failing the test, to avoid teeeedious repetition of is(!is.na(my.complicated.expression) & my.complicated.expression) .... They are deliberately not vectorized (contrary to some versions of mvbutils documentation); arguments with non-1 length trigger a warning.

legal.filename coerces its character argument into a similar-looking string that is a legal filename on any (?) system.

lsall is like ls but coerces all.names=TRUE.

masked checks which objects in search()[pos] are masked by identically-named objects higher in the search path. masking checks for objects mask identically-named objects lower in the search path. Namespaces may make the results irrelevant.

mkdir makes directories; unlike dir.create, it can do several levels at once.

most.recent returns the highest-so-far position of TRUE within a logical vector, or 0 if TRUE has not occurred yet; most.recent(c(F,T,F,T)) returns c(0,2,2,4).

my.all.equal is like all.equal, except that it returns FALSE in cases where all.equal returns a non-logical-mode result.

named(x) is just names(x) <- as.character(x); x; useful for lapply etc.

nscat, nscatn: see scatn

option.or.default obsolete— use equivalent getOption() instead.

pos is probably to be eschewed in new code, in favour of gregexpr with fixed=TRUE, which is likely faster. (And I should rewrite it to use gregexpr.) It's one of a few legacy functions in mvbutils that pre-date improvements in base R. pos will either search for several literal patterns in a single target, or vice versa– but not both. It returns a matrix showing the positions of the matching substrings, with as many columns as the maximum number of matches. 0 signifies "no match"; there is always at least one column even if there are no matches at all.

returnList returns a list corresponding to old-style (pre-R 1.8) return syntax. Briefly: a single argument is returned as itself. Multiple arguments are returned in a list. The names of that list are the argument names if provided; or, for any unnamed argument that is just a symbolic name, that symbolic name; or no name at all, for other unnamed arguments. You can duplicate pre-1.8 behaviour of return(...) via return(returnList(...)).

safe.rbind (*Deprecated in 2013*) mimics rbind, but works round an R bug (I reckon) where a column appears to be a numeric in one data.frame but a factor in the other. But I now think you should just sort your column classes/types properly in advance, rather than mixing types and relying on somewhat arbitrary conversion rules.

scatn is just cat(sprintf(fmt, ...), "", file=file, sep=sep). scatn prints a newline afterwards, but not before; nscat does the opposite; nscatn does both. If you're just displaying a "title" before calling print, use nscat.

to.regexpr converts literal strings to their equivalent regexps, e.g. by doubling backslashes. Useful if you want "fixed=TRUE" to apply only to a portion of your regexp.

yes.no cats its "prompt" argument and waits for user input. if the user input pmatches "yes" or "YES", then yes.no returns TRUE; if the input pmatches no or NO then yes.no returns FALSE; if the input is " and default is set, then yes.no returns default; otherwise it repeats the question. You probably want to put a space at the end of prompt.

Value

as.cat	character vector of class cat
clip	vector of the same mode as x
cq	character vector
empty.data.frame	data.frame
env.name.string	a string
expanded.call	a call object
everyth	same type as x
find.funs	a character vector of function names
find.lurking.envs	a data.frame with columns "what" and "size"
integ	scalar
is.dir	logical vector
is.nonzero	TRUE or FALSE
isF, isT	TRUE or FALSE
legal.filename	character(1)
masked	character vector
masking	character vector
mkdir	logical vector of success/failure
nscat	NULL
nscatn	NULL
most.recent	integer vector the same length as lvec, with values in the range (0,length(lvec)).
named	vector of the same mode as x
option.or.default	option's value
pos	numeric matrix, one column per match found plus one; at least one column guaranteed
returnList	list or single object
safe.rbind	data.frame
scatn	NULL
to.regexpr	character
yes.no	TRUE or FALSE

Arguments by function

as.cat x: character vector that you want to be displayed via `cat( x, sep="\n")`

clip x: a vector or list

clip n: integer saying how many elements to clip from the end of `x`

cq ...: quoted or unquoted character strings, to be `substituted` and then concatenated

deparse.names.parsably x: any object for `deparse`- `name` objects treated specially

empty.data.frame ...: named length-1 vectors of appropriate mode, e.g. "first.col=’"

env.name.string env: environment

expanded.call nlocal: frame to retrieve arguments from. Normally, use the default; see `mlocal`.

everyth x: subsettable thing. by: step between values to extract. from: first position.

find.funs ...: extra arguments for `objects`. Usually just "pattern" for regexp searches.

find.funs exclude.mcache: if TRUE (default), don’t look at `mlazy` objects

find.funs mode: "function" to look for functions, "environment" to look for environments, etc

find.lurking.envs delve: whether to recurse into function arguments and function bodies

find.lurking.envs trace: just a debugging aid– leave as FALSE

index lvector: vector of TRUE/FALSE/NA

integ expr: an expression; what: a string, the argument of `expr` to be integrated over; lo, hi: limits; ...: other variables to be set in the expression; args.to.integrate: a list of other things to pass to `integrate`

is.dir dir: character vector of files to check existence and directoriness of.

isF, isT x: anything, but meant to be a logical scalar

legal.filename name: character string to be modified

find.funs pos: list of environments, or vector of char or numeric positions in search path.

lsall ...: as for `ls`, except that `all.names` will be coerced to TRUE

masking, masked pos: position in search path

mkdir dirlist: character vector of directories to create

most.recent logical vector

my.all.equal x, y: anything; ...: passed to `all.equal`

named x: character vector which will become its own `names` attribute

nscat, nscatn see `scatn`

option.or.default opt.name: character(1)

option.or.default default: value to be returned if there is no `option` called `"opt.name"`

pos substrs: character vector of patterns (literal not regexpr)

pos mainstrs: character vector to search for `substrs` in.

pos any.case: logical- ignore case?

pos names.for.output: character vector to label rows of output matrix; optional

put.in.session ...: a named set of objects, to be `assigned` into the `mvb.session.info` search environment

returnList ...: named or un-named arguments, just as for `return` before R 1.8.

safe.rbind df1, df2: `data.frame` or `list`

scatn, nscat fmt, ...: as per `sprintf`; file, sep: as per `cat`

to.regexpr x: character vector

yes.no prompt: string to put before asking for input

yes.no default: value to return if user just presses <ENTER>

Author(s)

Mark Bravington

Examples

```
# as.cat
ugly.bugly <- c( 'A rose by any other name', 'would annoy taxonomists')
ugly.bugly
#[1] "A rose by any other name"                 "would annoy taxonomists"
as.cat( ugly.bugly) # calls print.cat--- no clutter
#A rose by any other name
#would annoy taxonomists
clip( 1:5, 2) # 1:3
cq( alpha, beta) # c( "alpha", "beta")
empty.data.frame( a=1, b="yes")
# data.frame with 0 rows of columns "a" (numeric) and "b" (character)
empty.data.frame( a=1, b=factor( c( "yes", "no")))$b
# factor with levels c( "no", "yes")
everyth( 1:10, 3, 5) # c( 5, 8)
f <- function( a=9, b) expanded.call(); f( 3, 4) # list( a=3, b=4)
find.funs( "package:base", patt="an") # "transform" etc.
find.lurking.envs( cd)
#                                     what  size
#1                     attr(obj, "source")  5368
#2                                  obj 49556
#3 environment(obj) <: namespace:mvbutils>   Inf
## Not run:
eapply( .GlobalEnv, find.lurking.envs)

## End(Not run)
integ( sin(x), 0, 1) # [1] 0.4597
integ( sin(x+a), a=5, 0, 1) # [1] -0.6765; 'a' is "passed" to 'expr'
integ( sin(y+a), what='y', 0, 1, a=0) # [1] 0.4597; arg is 'y' not 'x'
is.dir( getwd()) # TRUE
isF( FALSE) # TRUE
isF( NA) # FALSE
isF( c( FALSE, FALSE)) # FALSE, with a warning
sapply( c( FALSE, NA, TRUE), isF)
# [1]  TRUE FALSE FALSE
sapply( c( FALSE, NA, TRUE), isT)
# [1] FALSE FALSE  TRUE
legal.filename( "a:b\\c/d&f") # "a.b.c.d&f"
most.recent( c( FALSE,TRUE,FALSE,TRUE)) # c( 0, 2, 2, 4)
```

```
sapply( named( cq( alpha, beta)), nchar)  # c( alpha=5, beta=4)
pos( cq( quick, lazy), "the quick brown fox jumped over the lazy dog")
# matrix( c( 5, 37), nrow=2)
pos( "quick", c( "first quick", "second quick quick", "third"))
# matrix( c( 7,8,0, 0,14,0), nrow=3)
pos( "quick", "slow") # matrix( 0)
f <- function() { a <- 9; return( returnList( a, a*a, a2=a+a)) }
f() # list( a=9, 81, a2=18)
scatn( 'Things %i', 1:3)
nscat( 'Things %i', 1:3)
nscatn( 'Things %i', 1:3)
to.regexpr( "a{{") # "a\\{\\{"
## Not run:
mkdir( "subdirectory.of.getwd")
yes.no( "OK (Y/N)? ")
masking( 1)
masked( 5)

## End(Not run)
```

`my.index` *Arbitrary-level retrieval from and modification of recursive objects*

Description

As of R 2.12, you probably don't need these at all. But, in case you do: `my.index` and `my.index.assign` are designed to replace `[[` and `[[<-` *within* a function, to allow arbitrary-depth access into any recursive object. In order to avoid conflicts with system usage and/or slowdowns, it is wise to do this only inside a function definition where they are needed. A zero-length index returns the entire object, which I think is more sensible than the default behaviour (chuck a tanty). `my.index.exists` tests whether the indexed element actually exists. Note that these functions were written in 2001; since then, base-R has extended the default behaviour of `[[` etc for recursive objects, so that `my.index( thing, 1, 3, 5)` can sometimes be achieved just by to `thing[[c(1,3,5)]]` with the system version of `[[`. However, at least as of R 2.10.1, the system versions still have limited "recursability".

Usage

```
# Use them like this, inside a function definition:
# assign( "[[", my.index); var[[i]]
# assign( "[[<-", my.index.assign); var[[i]] <- value
my.index( var, ...) # not normally called by name
my.index.assign( var, ..., value) # not normally called by name
my.index.exists( i, var)
```

Arguments

`var`	a recursive object of any mode (not just `list`, but e.g. `call` too)
`value`	anything

...	one or more numeric index vectors, to be concatenated
i	numeric index vector

Details

Although R allows arbitrary-level access to lists, this does not (yet) extend to `call` objects or certain other language objects– hence these functions. They are written entirely in R, and are probably very slow as a result. Notwithstanding EXAMPLES below, it is **unwise** to replace system `[[` and `[[<-` with these replacements at a global level, i.e. outside the body of a function– these replacements do not dispatch based on object class, for example.

Note that `my.index` and `my.index.assign` distort strict R syntax, by concatenating their `...` arguments before lookup. Strictly speaking, R says that `x[[2,1]]` should extract one element from a matrix list; however, this doesn't really seem useful because the same result can always be achieved by `x[2,1][[1]]`. With `my.index`, `x[[2,1]]` is the same as `x[[c(2,1)]]`. The convenience of automatic concatentation seemed slightly preferable (at least when I wrote these, in 2001).

`my.index.exists` checks whether `var` is "deep enough" for `var[[i]]` to work. Unlike the others, it does not automatically concatenate indices.

At present, there is no facility to use a mixture of character and numeric indexes, which you can in S+ via "list subscripting of lists".

Author(s)

Mark Bravington

Examples

```
local({
  assign( "[[", my.index)
  assign( "[[<-", my.index.assign)
  ff <- function() { a <- b + c }
  body( ff)[[2,3,2]] # as.name( "b")
  my.index.exists( c(2,3,2), body( ff)) # TRUE
  my.index.exists( c(2,3,2,1), body( ff)) # FALSE
  body( ff)[[2,3,2]] <- quote( ifelse( a>1,2,3))
  ff # function () { a <- ifelse(a > 1, 2, 3) + c }
  my.index.exists( c(2,3,2,1), body( ff)) # now TRUE
})
```

NEG *Generate a negated version of your function. Useful for 'nlminb' etc.*

Description

You pass it a function `f(.)`; it returns a function whose result will be `-f(.)`. The arguments, return attributes, and environment are identical to those of `f`.

Usage

```
NEG(f)
```

Arguments

f	Normally, a function that returns a scalar; rarely, a NULL.

Value

A function that returns -f. However, if is.null(f), the result is also NULL; this is useful e.g. for gradient arg to nlminb.

Examples

```
NEG( sqrt)( 4) # -2
# should put in more complex one here...
e <- new.env()
e$const <- 3
funco <- function( x) -sum( ( x-const)^2L)
environment( funco) <- e
nlminb( c( 0, 0), NEG( funco)) # c( 3, 3)
dfunco <- NULL
nlminb( c( 0, 0), NEG( funco), gradient=NEG( dfunco)) # c( 3, 3)
```

---

noice *Prints a call object nicely*

Description

Prints a call-mode object nicely, with one argument per line. This is useful, for example, in displaying readably the outcomes of sys.call(), which is often used to create a call attribute for the results of complicated functions.

Usage

```
noice( cc, ...)
```

Arguments

cc	a call object, eg something appended to a fitting result via sys.call.
...	any other arguments for deparse

Value

Character vector with one argument per line, of class as.cat so that it prints cleanly. Long arguments are truncated, so the result is not guaranteed to re-parse cleanly (a general issue with R which seems unavoidable in any powerful language).

Examples

```
# This is a bona fide function call from my own work
# normally it would be evaluated directly, and sys.call()
# would be used inside it to assign a 'call' attrib to the result
# but the call attrib then looks like a mess-o-rama
# The quote() wrapper is just used here to make the point
# It would be interesting if 'call' could cope with a 'source' or
# 'srcref' argument, and would "know" how to print itself, but that
# is a big ask
# BTW, the 72-char limit in Rd EXAMPLES and USAGE is a PITBA
monster <- quote( est_N(
    popcompo = fp1a_17,
    df_rs_as_at_l = NULL,
    df_rs_ls = NULL, # NB comments are allowed, but get chucked
    newstyle_data = data17b,
    use_alpha_hsp = TRUE,
    AMIN = 8, AMAX = 30,
    YMIN = 2002, YMAX = 2014,
    prior_mean_z_plusgroup = 0.386,
    prior_sd_z_plusgroup = 0.0268,
    LMIN = 150, LMAX = 200,
    logit_surv_form = ~ I( pmax( age, 19)- AMAX) - 1,
    log_nsa_y1_form = ~factor(sex),
    log_nys_a1_reqm_form = ~0,
    logit_tresid_form = ~sex * I(len - 170),
    log_selbase_form = ~ 0,
    log_daily_reprodm_form = ~ 0,
    vb_form = ~sex,
    log_vb_cv_Linf_form = ~1,
    log_rct_re_var_start = log( sqr( 0.41)),
    fix_CV_R = TRUE,
    RE_rct = TRUE,
    sel_is_by_sex = TRUE,
    ssreduce_l = 1,
    fec_bout_start_fit = start_of_bout,
    fec_rest_start_fit = start_of_rest,
    fec_ovwt_fit = bfec,
    lf_sel_model=lv10kk5fix,
    nu_lata = 12))
monster # yuk
noice( monster) # yum
```

`pre.install` *Update a source and/or installed package from a task package*

Description

See `mvbutils.packaging.tools` before reading or experimenting!

pre.install creates a "source package" from a "task package", ready for first-time installation using install.pkg. You must have called maintain.packages(mypack) at some point in your R session before pre.install(mypack) etc.

patch.install is normally sufficient for subsequent maintenance of an already-installed package (ie you rarely need call install.pkg again). Again, maintain.packages must have been called earlier. It's also expected that the package has been loaded via library() before patch.install is called, but this may not be required. patch.install first calls pre.install and then modifies the installed package accordingly on-the-fly, so there is no need to re-load or re-build or re-install. patch.install also updates the help system with immediate effect, i.e. during the current R session. You don't need to call patch.install after every little maintenance change to your package during an R session; it's usually only necessary when (i) you want updated help, or (ii) you want to make the changes "permanent". However, it's not a problem to call patch.install quite often. patch.installed is a synonym for patch.install.

It's possible to tweak the source-package-creation process, and this is what 'pre.install.hook..." is for; see **Details** and section on **Overriding defaults** below.

spkg is a rarely-needed utility that returns the folder of source package created by pre.install.

Usage

```
# 95% of the time you just need:
# pre.install( pkg)
# patch.install( pkg)
# Your own hook: pre.install.hook.<<mypack>>( default.list, <<myspecialargs>>, ...)
pre.install( pkg, character.only=FALSE, force.all.docs=FALSE,
    dir.above.source="+", autoversion=getOption("mvb.autoversion", TRUE),
    R.target.version=getRversion(), ...)
patch.installed( pkg, character.only=FALSE, force.all.docs=FALSE,
    help.patch=TRUE, DLLs.only=FALSE,
    update.installed.cache=getOption("mvb.update.installed.cache", TRUE),
    pre.inst=!DLLs.only, dir.above.source="+", R.target.version=getRversion(),
    autoversion=getOption("mvb.autoversion", TRUE))
patch.install(...) # actually, args are exactly as for 'patch.installed'
spkg( pkg)
```

Arguments

pkg	package name. Either quoted or unquoted is OK; unquoted will be treated as quoted unless character.only=TRUE. Here and in most other places in mvbutils, you can also specify an actual in-memory-task-package object such as ..mypack.
character.only	Default FALSE, which allows unquoted package names. You can set it to TRUE, or just set e.g. char="my@funny@name", which will trump any use of pkg.
force.all.docs	normally just create help files for objects whose documentation has changed (which will always be generated, regardless of force.all.docs). If TRUE, then recreate help for all documented objects. Can also be a character vector of specific docfile names (usually function names, but can be the names of the

Rd file, without path or the Rd extension), in which case those Rd files will be regenerated.

`help.patch` if TRUE, patch the help of the installed package

`DLLs.only` just synchronize the DLLs and don't bother with other steps (see **Compiled code**)

`default.list` list of various things– see under "Overriding..." below

`...` arguments to pass to your `pre.install.hook.XXX` function, usually if you want to be able to build different "flavours" of a package (e.g. a trial version vs. a production version, or versions with and without enormous datasets included). In `patch.install`, `...` is just shorthand for the arg list of `patch.installed`.

`update.installed.cache`
If TRUE, then clear the installed-package cache, so that things like `installed.packages` work OK. The only reason to set to FALSE could be speed, if you have lots of packages; feedback appreciated. Default is TRUE unless you have set `options( mvb.update.installed.cache=FALSE)`.

`pre.inst` ?run `pre.install` first? Default is TRUE unless `DLLs.only=TRUE`; leave it unless you know better.

`autoversion` if TRUE, try to automatically increment the version number in the source (and installed, if `patch.install`) packages; this means you don't have to change the DESCRIPTION object or file. However, if you have changed the DESCRIPTION object or file's version to something beyond the source/installed version, the larger number will take precedence; hence, you can force a "major" revision by manually increasing the 1st or 2nd component of the version in **Description**. Only versions with at least 3 levels will be updated:so 1.0.0 will go to 1.0.1, 1.0.0.0 will go to 1.0.0.1, but 1.0 will stay the same. Default is TRUE unless you have set `options( mvb.autoversion=FALSE)`.

`dir.above.source`
folder within which the source package will go, with a + at the start being shorthand for the task package folder (the default). Hence `pre.install( pkg=mypack, dir="+/holder")` will lead to creation of "holder/mypack" below the task folder of `mypack`. Set this manually if you have to maintain different versions of the package for different R versions, or different flavours of the package for other reasons, or if your source package must live in a "subversion tree" (whatever that is).

`R.target.version`
Not needed 99% of the time; use only if you want to create source package for a different version of R. Supercedes the `Rd.version` argument of `pre.install` pre-'mvbutils' 2.5.57, used to control the documentation format. Set `R.target.version` to something less than "2.10" for ye olde "Rd version 1" format.

Details

As per the Glossary section of `mvbutils.packaging.tools`: the "task package" is the directory containing the ".RData" file with the guts of your package, which should be linked into the `cd` task hierarchy. The "source package" is usually the directory "«pkg»" below the task package, which will be created if needs be.

The default behaviour of pre.install is as follows– to change it, see **Overriding defaults**. A basic source package is created in a sourcedirectory "«pkg»" of the current task. The package will have at least a DESCRIPTION file, a NAMESPACE file, a single R source file with name "«pkg».R" in the "R" sourcedirectory, possibly a "sysdata.rda" file in the same place to contain non-functions, and a set of Rd files in the "man" sourcedirectory. Rd files will be auto-created from flatdoc style documentation, although precedence will be given to any pre-existing Rd files found in an "Rd" sourcedirectory of your task, which get copied directly into the package. Any "inst", "demo", "vignettes", "tests", "src", "exec", and "data" sourcedirectories will be copied to the source package, recursively (i.e. including any of *their* sourcedirectories). There is no compilation of source code, since only a source package is being created; see also **Compiled code** below.

Most objects in the task package will go into the source package, but there are usually a few you wouldn't want there: objects that are concerned only with how to create the package in the first place, and ephemeral system clutter such as .Random.seed. The default exceptions are: functions pre.install.hook.<<pkg>>, .First.task, and .Last.task; data <<pkg>>.file.exclude.regexes, <<pkg>>.DESCRIPTION, <<pkg>>.VERSION, <<pkg>>.UNSTABLE, forced!exports, .required, .Depends, tasks, .Traceback, .packageName, last.warning, .Last.value, .Random.seed, .SavedPlots; and any character vector whose name ends with ".doc".

All pre-existing files in the "man", "src", "tests", "exec", "demo", "inst", and "R" sourcedirectories of the source-package directory will be removed (unless you have some mlazy objects; see below). If a file ".Rbuildignore" file is present in the task package, it's copied to the package directory, but I've never gotten this feature to work (NB I should include a facility in the pre-install hook for this). To exclude files that would otherwise be copied, i.e. those in "inst/demo/src/data" folders, create a character vector of regexes called <<pkg>>.file.exclude.regexes; any file matching any of these won't be copied. If there is a "changes.txt" file in the task package, it will be copied to the "inst" sourcedirectory of the package, as will any files in the task's own "inst" sourcedirectory. A DESCRIPTION file will be created, preferably from a <<pkg>>.DESCRIPTION *object* in the task package; see mvbutils.packaging.tools for more. Any "Makefile.*" in the task package will be copied, as will any in the "src" sourcedirectory (not sure why both places are allowed). No other files or sourcedirectories in the package directory will be created or removed, but some essential files will be modified.

If a NAMESPACE file is present in the task (usually no need), then it is copied directly to the package. If not, then pre.install will generate a NAMESPACE file by calling make.NAMESPACE, which makes reasonable guesses about what to import, export, and S3methodize. What is & isn't an S3 method is generally deduced OK (see make.NAMESPACE for gruesome details), but you can override the defaults via the pre-install hook. FWIW, since adding the package-creation features to mvbutils, I have never bothered explicitly writing a NAMESPACE file for any of my packages. By default, only *documented* functions are exported (i.e. visible to the user or other packages); the rest are only available to other functions in your package.

The R source file will contain functions only. Any doc and export.me attributes are dropped, but other attributes are kept; in particular, source code is kept in the source attribute.

If any of the Rd files starts with a period, e.g. ".dotty.name", it will be renamed to e.g. "01.dotty.name.Rd" to avoid some problems with RCMD. This should never matter, but just so you know...

To speed up conversion of documentation, a list of raw & converted documentation is stored in the file "doc2Rd.info.rda" in the task package, and conversion is only done for objects whose raw documentation has changed, unless force.all.docs is TRUE.

pre.install creates a file "funs.rda" in the package's "R" sourcedirectory, which is subsequently

used by patch.installed. The function build.pkg (or R CMD BUILD) and friends will omit this file (currently with a complaint, which I intend to fix eventually, but which does not cause trouble).

Compiled code: patch.install does not compile source code; currently, you need to do that yourself, though I might add support for that if I can work a sufficiently general mechanism. If you use R to do your compilation, then install.pkg should work after pre.install, though you may need detach("package:mypack", unload=T) first and that will disrupt your R session. Alternatively, you may be able to use R CMD SHLIB to create the DLL directly, which you can then copy into the "libs" sourcedirectory of the installed package, without needing to re-install. I haven't tried this, but colleagues have reported success.

If, like me, you pre-compile your own DLLs manually (not allowed on CRAN, but fine for distribution to other users on the same OS), then you can put the DLLs into a folder "inst/libs" of the task (see next for Windows); they will end up as usual in the "libs" folder of the installed package, even though R itself hasn't compiled them. On Windows, put the DLLs one level deeper in "inst/libs/«arch»" instead, where "«arch»" is found from .Platform$r_arch; for 32-bit Windows, it's currently "i386". All references in this section to "libs", whether in the task or source or installed package, should be taken as meaning "libs/«arch»".

To load your package's DLLs, call library.dynam in the .onLoad function, for example like this:

```
.onLoad <- function( libname, pkgname){
  library.dynam( 'my_first_dll', package=pkgname)
 library.dynam( 'my_other_dll', package=pkgname)  # fine to have several DLLs
}
```

To automatically load all DLLs, you can copy the body of mvbutils:::generic.dll.loader into your own .onLoad, or just include a call to generic.dll.loader(libname,pkgname) if you don't mind having dependence on mvbutils.

After the package has been installed for the first time, I change my compiler settings so that the DLL is created directly in the installed package's "libs" folder; this means I can use the compiler's debugger while R is running. To accommodate this, patch.install behaves as follows:

- any new DLLs in the task package are copied to the installed package;
- any DLLs in the installed package but not in the task package are deleted;
- for any DLLs in both task & installed, both copies are synchronized to the *newer* version;
- the source package always matches the task package

You can call patch.install(mypack, DLLs.only=TRUE) if you only want the DLL-synching step.

(Before version 2.5.57, mvbutils allowed more latitude in where you could put your home-brewed DLLs, but it just made life more confusing. The only place that now works is as above.)

Data objects: Data objects are handled a bit differently to the recommendations in "R extensions" and elsewhere– but the end result for the package user is the same, or better. The changes have been made to speed up package maintenance, and to improve useability. Specifically:

- Undocumented data objects live only in the package's namespace, i.e. visible only to your functions.
- Documented data objects appear both in the visible part of the package (i.e. in the search path), and in the namespace. [The R standard is that these should not be visible in the namespace, but this doesn't seem sensible to me.]

- The easiest way to export a data object, is to "document" it by putting its name into an alias line of the doc attribute of an existing function. (Alias lines are single-word lines directly after the first line of the doc attr.)
- To document a data object xxx in its own right, include a flat-format text object xxx.doc in your task package; see doc2Rd. xxx.doc itself won't appear in the packaged object, but will result in documentation for xxx *and* any other data objects that are given as alias lines.
- Big data objects can be set up for transparent individual lazy-loading (see below) to save time & memory, but lazy-loading is otherwise off by default for individual data objects.
- There is no need for the user ever to call data to access a dataset in the package, and in fact it won't work.

Note that the data(...) function has been pretty much obsolete since the advent of lazy-loading in R 2.0; see R-news #4/2.

In terms of package structure, as opposed to operation, there is no "data" sourcedirectory. Data lives either in the "sysdata.rdb/rdx" files in the "R" sourcedirectory (but can still be user-visible, which is not normally the case for objects in those files), or in the "mlazy" sourcedirectory for those objects with individual lazy-loading.

Big data objects: Lazy-loading objects cached with mlazy are handled specially, to speed up pre.install. Such objects get their cache-files copied to "inst/mlazy", and the .onLoad is prepended with code that will load them on demand. By default, they are exported if and only if documented, and are not locked. The following objects are not packaged by default, even if mlazyed: .Random.seed, .Traceback, last.warning, and .Saved.plots. These are mlazyed automatically if options(mvb.quick.cd) is TRUE– see cd.

Documentation and exporting:

Package documentation: Just because you have a package **Splendid**, it doesn't follow that a user will be able to figure out how to use it from the alphabetical list of functions in library(help=Splendid); even if you've written vignettes, it may not be obvious which to use. The recommended way to provide a package overview is via "package documentation", which the user accesses via package?Splendid. You can write this in a text object called e.g. "Splendid.package.doc", which will be passed through doc2Rd with an extra "docType{package}" field added. The first line should start e.g. "Splendid-package" and the corresponding ".Rd" file will be put first into the index. Speaking as a frequently bewildered would-be user of others' packages– and one who readily gives up if the "help" is impenetrable– I urge you to make use of this feature!

Vignettes: See mvbutils.packaging.tools.

Bare minimum for export: Only documented functions and data are exported from your package (unless you resort to the subterfuge described in the subsection after this). Documented things are those found by find.documented(doc="any"). The simplest way to document something is just to add its name as an "alias line" to the existing documentation of another function, before the first empty line. For example, if you're already using flatdoc to document my.beautiful.function, you can technically "document" and thus export other functions like so:

```
structure( function( blahblahblah)...
,doc=flatdoc())
my.beautiful.function    package:splendid
other.exported.function.1
other.exported.function.2
```

The package will build & install OK even if you don't provide USAGE and ARGUMENTS sections for the other functions. Of course, R CMD CHECK wouldn't like it (and may have a point on this occasion). If you just are after "legal" (for R CMD CHECK) albeit unhelpful documentation for some of your functions that you can't face writing proper doco for yet, see `make.usage.section` and `make.argument.section`.

Exporting undocumented things and vice versa: A bit naughty (RCMD CHECK complains), but quite doable. Note that "things" can be data objects, not just functions. Simply write a pre-install hook (see **Overriding defaults**) that includes something like this:

```
pre.install.hook.mypack <- function( hooklist) {
 hooklist$nsinfo$exports <- c( hooklist$nsinfo$exports, "my.undocumented.thing")
return( hooklist)
}
```

You can follow a similar approach if you want to document something but *not* to export it (so that it can only be accessed by `Splendid:::unexported.thing`. This probably isn't naughty.

Overriding defaults: Source package folder can be controlled via `options("mvbutils.sourcepkgdir.postfix")`, as per "Folders and different R versions" in `mvbutils.packaging.tools`. You'd only need to do this if you have multiple R versions installed that require different source-package formats (something that does not often change).

If a function `pre.install.hook.<<pkgname>>` exists in the task "«pkgname»", it will be called during `pre.install`. It will be passed one list-mode argument, containing default values for various installation things that can be adjusted; and it should return a list with the same names. It will also be passed any `...` arguments to `pre.install`, which can be used e.g. to set "production mode" vs "informal mode" of the end product. For example, you might call `preinstall(mypack,modo="production")` and then write a function `pre.install.hook.mypack( hooklist, modo)` that includes or excludes certain files depending on the value of `modo`. The hook can do two things: sort out any file issues not adequately handled by `pre.install`, and/or change the following elements in the list that is passed in. The return value should be the possibly-modified list. Hook list elements are:

copies files to copy directly

dll.paths DLLs to copy directly

extra.filecontents named list; each element is the contents of a text file, the corresponding name being the path of the file to create eg `"inst/src/utils.pas"`— a nonstandard name

extra.docs names of character-mode objects that constitute flat-format documentation

description named elements of DESCRIPTION file

task.path path of task (ready-to-install package will be created as a sourcedirectory in this)

has.namespace should a namespace be used?

use.existing.NAMESPACE ignore default and just copy the existing NAMESPACE file?

nsinfo default namespace information, to be written iff `has.namespace==TRUE` and `use.existing.NAMESPACE==FALSE`

exclude.funs any functions **not** to include

exclude.data non-functions to exclude from `system.rda`

dont.check.visibility either TRUE (default default), FALSE, or a specified character vector, to say which objects are *not* to be checked for "globality" by RCMD CHECK (using the `globalVariables` mechanism). Leave alone if you don't understand this. You can change the "default default" via `options( mvb_dont_check_visibility=FALSE)`.

There are two reasons for using a hook rather than directly setting parameters in `pre.install`. The first is that `pre.install` will calculate sensible but non-obvious default values for most

things, and it is easier to change the defaults than to set them up from scratch in the call. The second is that once you have written a hook, you can forget about it– you don't have to remember special argument values each time you call `pre.install` for that task.

Debugging a pre install hook: To understand what's in the list and how to write a pre-install hook, the easiest way is probably to write a dummy one and then `mtrace` it before calling `pre.install(mypack)`. However, it's all a bit clunky at present (July 2011). Because the hook only exists in the "..mypack" shadow environment, `mtrace` won't find it automatically, so you'll need `mtrace( pre.install.hook.mypack, from=.mypack)`. That's fine, but if you then modify the source of your hook function, you'll get an error following the "Reapplying trace..." message. So you need to do `mtrace.off` *before* saving your edited hook-function source, and then `mtrace` the hook again before calling `pre.install(mypack)`. To be fixed, if I can work out how...

Different versions of r: R seems to be rather fond of changing the structural requirements of source & installed packages. `mvbutils` tries to shield you from those arcane and ephemeral details– usually, your task package will not need changing, and `pre.install` will automatically generate source & installed packages in whatever format R currently requires. However, sometimes you do at least need to be able to build different "instances" of your package for different versions of R. The `sourcedir` and maybe the `R.target.version` arguments of `pre.install` may help with this.

But if you need to build instances of your package for a different version of R, then you may need this argument (and `dir.above.source`). I try to keep `mvbutils` up-to-date with R's fairly frequent revisions to package structure rules, with the aim that you (or I) can easily produce a source/binary-source package for a version of R later than the one you're using right now, merely by setting `R.target.version`. However, be warned that this may not always be enough; there might at some point be changes in R that will require you to be running the appropriate R version (and an appropriate version of `mvbutils`) just to recreate/rebuild your package in an appropriate form.

The nuances of `R.target.version` change with the changing tides of R versions, but the whole point of `pre.install` etc is that you shouldn't really need to know about those details; `mvbutils` tries to look after them for you. For example, though: as of 10/2011, the "detailed behaviour" is to enforce namespaces if `R.target.version` >= 2.14, regardless of whether your package has a `.onLoad` or not.

Packages without namespaces pre r2 14: You used to be allowed to build packages without namespaces– not to be encouraged for general distribution IMO, but occasionally a useful shortcut for your own stuff nevertheless (mainly because everything is "exported", documented or not). For R <= 2.14, `mvbutils` will decide for itself whether your package is meant to be namespaced, based on whether any of the following apply: there is a NAMESPACE file in the task package; there is a `.onLoad` function in the task; there is an "Imports" directive in the DESCRIPTION file.

Author(s)

Mark Bravington

See Also

`mvbutils.packaging.tools`, `cd`, `doc2Rd`, `maintain.packages`

Examples

```
## Not run:
# Workflow for simple case:
cd( task.above.mypack)
maintain.packages( mypack)
# First-time setup, or after major R version changes:
pre.install( mypack)
install.pkg( mypack)
library( mypack)
# ... do stuff
# Subsequent maintenance:
maintain.packages( mypack) # only once per session, usually at the start
library( mypack) # maybe optional
# ...do various things involving changes to mypack, then...
patch.install( mypack) # keep disk image up-to-date
# Prepare copies for distribution
build.pkg( mypack) # for Linux or CRAN
build.pkg.binary( mypack) # for Windows or Macs
check.pkg( mypack) # if you like that sort of thing

## End(Not run)
```

`print` *Print values*

Description

See base-R documentation of `print` and `print.default`. Users should see no difference with the `mvbutils` versions; they need to be documented and exported in `mvbutils` for purely technical reasons. There are also three useful special-purpose print methods in `mvbutils`; see **Value**.Some of the base-R documentation is reproduced below.

The motive for redeclaration is to have a seamless transition within the `fixr` editing system, from the nice simple "source"-attribute system used to store function source-code before R2.14, to the quite extraordinarily complicated "srcref" system used thereafter. `mvbutils` does so via an augmented version of base-R's print method for functions, without which your careful formatting and commenting would be lost. If a function has a "source" attribute but no "srcref" attribute (as would be the case for many functions created prior to R2.14), then the augmented `print.function` will use the "source" attribute. There is no difference from base-R in normal use.

See **How to override an s3 method** if you really want to understand the technicalities.

Usage

```
print(x, ...) # generic
## Default S3 method:
print(x, ...) # S3 method for default
## S3 method for class 'function'
print(x, useSource=TRUE, ...) # S3 method for function
```

```
## S3 method for class 'cat'
print(x, ...) # S3 method for cat
## S3 method for class 'specialprint'
print(x, ...) # S3 method for specialprint
## S3 method for class 'pagertemp'
print(x, ...) # S3 method for pagertemp
## S3 method for class 'call'
print(x, ...) # S3 method for call
## S3 replacement method for class '`<-`'
print(x, ...) # S3 method for "<-" (a special sort of call)
## S3 method for class '`(`'
print(x, ...) # S3 method for "(" (a special sort of call)
#print(x, ...) # S3 method for "{" (a special sort of call)
## S3 method for class '`if`'
print(x, ...) # S3 method for "if" (a special sort of call)
## S3 method for class '`for`'
print(x, ...) # S3 method for "for" (a special sort of call)
## S3 method for class '`while`'
print(x, ...) # S3 method for "while" (a special sort of call)
## S3 method for class 'name'
print(x, ...) # S3 method for name (symbol)
```

Arguments

x	thing to print.
...	other arguments passed to NextMethod and/or ignored. There are many special arguments to base-R print.default, as described in its documentation. They are not named individually in the mvbutils version for technical reasons, but you can still use them.
useSource	[print.function] logical, indicating whether to use source references or copies rather than deparsing language objects. The default is to use the original source if it is available. The mvbutils override will print a "source" attribute if one exists but no "srcref" attribute does, whereas base-R post-2.14 would just print a deparsed version of the function.

Value

Technically, an invisible version of the object is returned. But the point of print is to display the object. print.function displays source code, as per **Description**. print.default and print.call need to exist in mvbutils only for technical reasons. The other two special methods are: print.cat applies to character vectors of S3 class cat, which are printed each on a new line, without the [1] prefix or double-quotes or backslashes. It's ideal for displaying "plain text". Use as.cat to coerce a character vector so that it prints this way. print.specialprint can be used to ensure an object (of class specialprint) displays in any particular way you want, without bothering to define a new S3 class and write a print method. Just give the object an attribute "print" of mode expression, which can refer to the main argument x and any other arguments. That expression will be run by print.specialprint– see **Examples**. print.pagertemp is meant only for internal use by the informal-help viewer.

How to override an s3 method

Suppose you maintain a package **mypack** in which you want to mildly redefine an existing S3 method, like `mvbutils` does with `print.function`. (Drastic redefinitions are likely to be a bad idea, but adding or tweaking functionality can occasionally make sense.) The aim is that other packages which import `mypack` should end up using your redefined method, and so should the user if they have explicitly called `library( mypack)`. But your redefined method should *not* be visible to packages that don't import `mypack`, nor to the user if `mypack` has only been loaded implicitly (i.e. if `mypack` is imported by another package, so that `asNamespace(mypack)` is loaded but `package:mypack` doesn't appear on the search path). It's hard to find out how to do this. Here's what I have discovered:

- For a *new* S3 method (i.e. for a class that doesn't already have one), then you just need to mark it as an `S3method` in the `mypack` NAMESPACE file (which `mvbutils` packaging tools do for you automatically). You don't need to document the new method explicitly, and consequently there's no need to export it. The new method will still be found when the generic runs on an object of the appropriate class.
- If you're modifying an existing method, you can't just declare it as `S3method` in the NAMESPACE file of `mypack`. If that's all you did, R would complain that it already has a registered method for that class— fair enough. Therefore, you also have to redeclare and export the *generic*, so that there's a "clean slate" for registering the method (specifically, in the S3 methods table for `mypack`, where the new generic lives). The new generic will probably be identical to the existing generic, very likely just a call to `UseMethod`. Because it's exported, it needs to be documented; you can either just refer to base-R documentation (but you still need all the formal stuff for Arguments etc, otherwise RCMD CHECK complains), or you can duplicate the base-R doco with a note. `help2flatdoc` is useful here, assuming you're wisely using `mvbutils` to build & maintain your package.
- If you redeclare the generic, you also need to make sure that your *method* is *exported* as well as S3-registered in the NAMESPACE file of `mypack`. This is somehow connected with the obscure scoping behaviour of `UseMethod` and I don't really understand it, but the result is: if you don't export your method, then it's not found by the new generic (even though it exists in `asNamespace(mypack)`, which is the environment of the new generic, and even though your method is also S3-registered in that same environment). Because you export the method, you also need to document it.
- Unfortunately, the new generic won't know about the methods already registered for the old generic. So, for most generics (exceptions listed later), you will also have to define a `generic.default` method in `mypack`— and you need to export and therefore document it too, as per the previous point. This `generic.default` just needs to invoke the original generic, so that the already-registered S3 methods are searched. However, this can lead to infinite loops if you're not careful. See `mvbutils:::print.default` for how to do it. If you were redefining a generic that was originally (or most recently) defined somewhere other than `baseenv()`, then you'd need to replace the latter with `asNamespace(<<original.defining.package>>)`.
- Because your new `generic.default` might invoke any of the pre-existing (or subsequently-registered) methods of the *original* generic, you should just make its argument list `x,...`. In other words, don't name individual arguments even if they are named in the original `generic.default` (eg for `print.default`).
- Objects of mode `name`, `call`, and `"("` or `"{"` or `"<-"` (special types of `call`) cause trouble in `generic.default` (at least using the approach in the previous point, as in `mvbutils:::print.default`).

Unless they have a specific method, the object will be automatically evaluated. So if your generic is ever likely to be invoked on a `call` object, you'll need a special `generic.call` method, as in `mvbutils:::print.call`; the same goes for those other objects.

- A few generics— `rbind` and `cbind`, for example— use their own internal dispatch mechanism and don't have e.g. an `rbind.default`. Of course, there is a default behaviour, but it's not defined by an R-level function; see `?InternalGenerics`. For these generics, the previous point wouldn't work as a way of looking for existing methods. Fortunately, at least for `rbind`, things seem to "just work" if your redefined generic simply runs the code of the base generic (but don't call the latter directly, or you risk infinite loops— just run its body code). Then, if *your* generic is run, the search order is (1) methods registered for *your* generic in `asNamespace("mypack")`, whether defined in `mypack` itself or subsequently registered by another package that uses `mypack`, (2) methods defined/registered for the base generic (ie in the original generic's namespace), (3) the original "implicit default method". But if the *original* generic is run (e.g. from another package that doesn't import `mypack`), then step (1) is skipped. This is good; if another package **pack2** imports `mypack` and registers an S3 method, the S3 registration will go into the `mypack` S3 lookup table, but if `pack2` *doesn't* import `mypack` then the S3 registration will go into the base S3 lookup table (or the lookup table for whichever package the generic was originally defined in, eg package **stats**).

Examples

```
## Not run:
# Special methods shown below; basic behaviour of 'print', 'print.default',
# and 'print.function' is as for base-R
#cat
ugly.bugly <- c( 'A rose by any other name', 'would annoy taxonomists')
ugly.bugly
#[1] "A rose by any other name"                "would annoy taxonomists"
as.cat( ugly.bugly) # calls print.cat--- no clutter
#A rose by any other name
#would annoy taxonomists
# nullprint:
biggo <- 1:1000
biggo
# [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
# [2] 19  20  21  22  23  24  25  26  27  28 etc...
oldClass( biggo) <- 'nullprint'
biggo # calls print.nullprint
# nuthin'
# specialprint:
x <- matrix( exp( seq( -20, 19, by=2)), 4, 5)
attr( x, 'print') <- expression( {
    x[] <- sprintf( '%12.2f', x);
    class( x) <- 'noquote';
    attr( x, 'print') <- NULL;
    print( x)
  })
class( x) <- 'specialprint'
x # calls print.specialprint; consistently formatted for once
#     [,1]         [,2]         [,3]         [,4]         [,5]
#[1,]         0.00         0.00         0.02        54.60    162754.79
```

```
#[2,]         0.00         0.00         0.14       403.43   1202604.28
#[3,]         0.00         0.00         1.00      2980.96   8886110.52
#[4,]         0.00         0.00         7.39     22026.47  65659969.14

## End(Not run)
```

---

rbdf *Data frames: better behaviour with zero-length cases*

Description

rbind concatenates its arguments by row; see cbind for basic documentation. There is an rbind method for data frames which mvbutils overrides, and rbdf calls the override directly. The mvbutils version should behave exactly as the base-R version, with two exceptions:

- zero-row arguments are **not** ignored, e.g. so that factor levels which never appear are not dropped.
- dimensioned (array or matrix) elements do not lose any extra attributes (such as class).

I find the zero-row behaviour more logical, and useful because e.g. it lets me create an empty.data.frame with the correct type/class/levels for all columns, then subsequently add rows to it. The behaviour for matrix (array) elements allows e.g. the rbinding of data frames that contain matrices of POSIXct elements without losing the POSIXct class (as in my package **nicetime**).

When rbinding data frames, best practice is to make sure all the arguments really are data frames. Lists and matrices also work OK (they are first coerced to data frames), but scalars are dangerous (even though base-R will process them without complaint). rbind is quirky around data frames; unless *all* the arguments are data frames, sometimes rbind.data.frame will not be called even when you'd expect it to be, and the coercion of scalars is frankly potty; see **Details** and EXAMPLES. mvbutils:::rbind.data.frame tries to mimic the base-R scalar coercion, but I'm not sure it's 100% compatible. Again, the safest way to ensure a predictable outcome, is to make sure all arguments really are data frames, and/or to call rbdf directly.

Note that ("thanks" to stringsAsFactors) the order in which data frames are rbound can affect the result— see **Examples**.

Obsolete: Versions of mvbutils prior to 2.8.207 installed replacements for $<-.data.frame and [[<-.data.frame that circumvented weird behaviour with the base-R versions when the data.frame had zero rows. That weird behaviour seems to be fixed in base-R as of version 3.4.4 (perhaps earlier). I've therefore removed those replacements (after warnings from newer versions RCMD CHECK). Hopefully, everything works... but just for the record, here's the old text, which I *think* no longer applies.

[I *think* this paragraph is obsolete.] Normally, you can replace elements in, or add a column to, data frames via e.g. x$y <- z or x[["y"]] <- z. However, in base-R this fails for no good reason if x is a zero-row data frame; the sensible behaviour when y doesn't exist yet, would be to create a zero-length column of the appropriate class. mvbutils overrides the base (mis)behaviour so it works sensibly. Should work for matrix/array "replacements" too.

Usage

```
rbind(..., deparse.level = 1) # generic
## S3 method for class 'data.frame'
rbind(..., deparse.level = 1) # S3 method for data.frame
rbdf(..., deparse.level = 1) # explicitly call S3 method...
# ... for data frames (circumvent rbind dispatch)
## OBSOLETE x[[i,j]] <- value # S3 method for data.frame; only ...
## OBS ... the version x[[i]] <- value is relevant here, tho' arguably j==0 might be
## OBS x$name <- value # S3 method for data.frame
```

Arguments

...	Data frames, or things that will coerced to data frames. NULLs are ignored.
deparse.level	not used by rbind.data.frame, it's for the default and generic only

Details

old arguments

i,j column and row subscripts

name column name

x, value that's up to you; I just have to include them here to stop RCMD CHECK from moaning... :/

See cbind documentation in base-R.

R's dispatch mechanism for rbind is as follows [my paraphrasing of base-R documentation]. Mostly, if any argument is a data frame then rbind.data.frame will be used. However, if one argument is a data frame but another argument is a scalar/matrix of a class that has an rbind method, then "default rbind" will be called instead. Although the latter still returns a data frame, it stuffs up e.g. class attributes, so that POSIXct objects will be turned into huge numbers. Again, if you really want a data frame result, make sure all the arguments are data frames.

In mvbutils:::rbind.data.frame (and AFAIK in the base-R version), arguments that are not data frames are coerced to data frames, by calling data.frame() on them. AFAICS this works predictably for list and matrix arguments; note that lists need names, and matrices need column names, that match the names of the real data frame arguments, because column alignment is done by name not position. Behaviour for scalars is IMO weird; see **Examples**. The idea seems to be to turn each scalar into a single-row data frame, coercing its names and truncating/replicating it to match the columns of the first real data frame argument; any names of the scalar itself are disregarded, and alignment is by position not name. Although mvbutils:::rbind.data.frame tries to mimic this coercion, it seems to me unnecessary (the user should just turn the scalar into something less ambiguous), confusing, and dangerous, so mvbutils issues a warning. Whether I have duplicated every quirk, I'm not sure.

Note also that R's accursed drop=TRUE default means that things you might reasonably think *should* be data frames, might not be. Under some circumstances, this might result in rbind.data.frame being bypassed. See **Examples**.

Short of rewriting data.frame and rbind, there's nothing mvbutils can do to fix these quirks. Whether base-R should consider any changes is another story, but back-compatibility probably suggests not.

Value

Taken from the base-R documentation, modified to fit the `mvbutils` version The `rbind` data frame method first drops any NULL arguments, then coerces all others to data frames (see **Details** for how it does this with scalars). Then it drops all zero-column arguments. (If that leaves none, it returns a zero-column zero-row data frame.) It then takes the classes of the columns from the first argument, and matches columns by name (rather than by position). Factors have their levels expanded as necessary (in the order of the levels of the levelsets of the factors encountered) and the result is an ordered factor if and only if all the components were ordered factors. (The last point differs from S-PLUS.) Old-style categories (integer vectors with levels) are promoted to factors. Zero-row arguments are kept, so that in particular their column classes and factor levels are taken account of. Because the class of each column is set by the *first* data frame, rather than "by consensus", numeric/character/factor conversions can be a bit surprising especially where NAs are involved. See the final bit of EXAMPLES.

See Also

`cbind` and `data.frame` in base-R; `empty.data.frame`

Examples

```
# mvbutils versions are used, unless base:: or baseenv() gets mentioned
# Why base-R dropping of zero rows is odd
rbind( data.frame( x='yes', y=1)[-1,], data.frame( x='no', y=0))$x # mvbutils
#[1] no
#Levels: yes no # two levels
base::rbind( data.frame( x='yes', y=1)[-1,], data.frame( x='no', y=0))$x # base-R
#[1] no
#Levels: no # lost level
rbind( data.frame( x='yes', y=1)[-1,], data.frame( x='no', y=0, stringsAsFactors=FALSE))$x
#[1] no
#Levels: yes no
base::rbind( data.frame( x='yes', y=1)[-1,], data.frame( x='no', y=0, stringsAsFactors=FALSE))$x
#[1] "no" # x has turned into a character
# Quirks of scalar coercion
evalq( rbind( data.frame( x=1), x=2, x=3), baseenv()) # OK I guess
#   x
#1  1
#x  2
#x1 3
evalq( rbind( data.frame( x=1), x=2:3), baseenv()) # NB lost element
#  x
#1 1
#x 2
evalq( rbind( data.frame( x=1, y=2, z=3), c( x=4, y=5)), baseenv())
# NB gained element! Try predicting z[2]...
#  x y z
#1 1 2 3
#2 4 5 4
evalq( rbind( data.frame( x='cat', y='dog'), cbind( x='flea', y='goat')), baseenv()) # OK
#     x    y
#1  cat  dog
```

```
#2 flea goat
evalq( rbind( data.frame( x='cat', y='dog'), c( x='flea', y='goat')), baseenv()) # Huh?
#Warning in `[<-.factor`(`*tmp*`, ri, value = "flea") :
#  invalid factor level, NAs generated
#Warning in `[<-.factor`(`*tmp*`, ri, value = "goat") :
#  invalid factor level, NAs generated
#     x    y
#1  cat  dog
#2 <NA> <NA>
evalq( rbind( data.frame( x='cat', y='dog'), c( x='flea')), baseenv()) # Hmmm...
#Warning in `[<-.factor`(`*tmp*`, ri, value = "flea") :
#  invalid factor level, NAs generated
#Warning in `[<-.factor`(`*tmp*`, ri, value = "flea") :
#  invalid factor level, NAs generated
#     x    y
#1  cat  dog
#2 <NA> <NA>
try( evalq( rbind( data.frame( x='cat', y='dog'), cbind( x='flea')), baseenv())) # ...mmmm...
#Error in rbind(deparse.level, ...) :
#  numbers of columns of arguments do not match
# Data frames that aren't:
data.frame( x=1,y=2)[-1,] # a zero-row DF-- OK
# [1] x y
# <0 rows> (or 0-length row.names)
data.frame( x=1)[-1,] # not a DF!?
# numeric(0)
data.frame( x=1)[-1,,drop=FALSE] # OK, but exceeeeeedingly cumbersome
# <0 rows> (or 0-length row.names)
# Implications for rbind:
rbind( data.frame( x='yes')[-1,], x='no')
#  [,1]
# x "no" # rbind.data.frame not called!
rbind( data.frame( x='yes')[-1,,drop=FALSE], x='no')
#Warning in rbind(deparse.level, ...) :
#  risky to supply scalar argument(s) to 'rbind.data.frame'
#   x
#x no
# Quirks of ordering and character/factor conversion:
rbind( data.frame( x=NA), data.frame( x='yes'))$x
#[1] NA    "yes" # character
rbind( data.frame( x=NA_character_), data.frame( x='yes'))$x
#[1] <NA> yes
#Levels: yes # factor!
rbind( data.frame( x='yes'), data.frame( x=NA))$x[2:1]
#[1] <NA>  yes
#Levels: yes # factor again
x1 <- data.frame( x='yes', stringsAsFactors=TRUE)
x2 <- data.frame( x='no', stringsAsFactors=FALSE)
rbind( x1, x2)$x
# [1] yes no
# Levels: yes no
rbind( x2, x1)$x
# [1] "no"  "yes"
```

```
# sigh...
```

`readLines.mvb` *Read text lines from a connection*

Description

Reads text lines from a connection (just like `readLines`), but optionally only until a specfied string is found.

Usage

```
readLines.mvb( con=stdin(), n=-1, ok=TRUE, EOF=as.character( NA), line.count=FALSE)
```

Arguments

`con`	A connection object or a character string.
`n`	integer. The (maximal) number of lines to read. Negative values indicate that one should read up to the end of the connection.
`ok`	logical. Is it OK to reach the end of the connection before 'n > 0' lines are read? If not, an error will be generated.
`EOF`	character. If the current line matches the EOF, it's treated as an end-of-file, and the read stops. The connection is left OPEN so that subsequent reads work.
`line.count`	(default FALSE) see **Value**.

Details

Apart from stopping if the EOF line is encountered, and as noted with `line.count==TRUE`, behaviour should be as for `readLines`.

Value

A character vector of length the number of lines read. If `line.count==TRUE`, it will also have an attribute "line.count" showing the number of lines read.

See Also

`source.mvb`, `current.source`, `flatdoc`

Examples

```
tt <- tempfile()
cat( letters[ 1:6], sep="\n", file=tt)
the.data <- readLines.mvb( tt, EOF="d")
unlink( tt)
the.data # [1] "a" "b" "c"
```

rm.pkg *Remove object(s) from maintained package*

Description

Remove object(s) from maintained package. If the package is loaded, then objects are also removed from the search path version if any, the namespace if any, any importing namespaces, and any S3 method table. `remove.from.package` is a synonym. You will be prompted about whether to auto-save the maintained package.

Usage

```
rm.pkg( pkg, ..., list = NULL, save.=NA)
# remove.from.package( pkg, ..., list=NULL)
remove.from.package( ...) # really has same args as 'rm.pkg'
```

Arguments

`pkg`	(string, or environment) package name or environment, e.g. `..mypack`
`...`	unquoted object names to remove
`list`	character vector alternative to ..., which is ignored if `list` is set
`save.`	For internal use— leave this alone!

Details

For now, methods are only removed from the **base** S3 methods table; if new S3 generics have been defined in loaded packages, and you are trying to remove a method for such a generic, then it won't be removed. I could implement this feature if anyone really wants it.

See Also

`maintain.packages`

Examples

```
## Not run:
rm.pkg( "mypackage", foo, bar)
rm.pkg( "mypackage", list=cq( foo, bar))
rm.pkg( ..mypackage, list=cq( foo, bar))

## End(Not run)
```

Save *Save R objects*

Description

These function resemble save and save.image, with two main differences. First, any functions which have been mtraced (see package **debug**) will be temporarily untraced during saving (the **debug** package need not be loaded). Second, Save and Save.pos know how to deal with lazy-loaded objects set up via mlazy. Save() is like save.image(), and also tries to call savehistory (see **Details**). Save.pos(i) saves all objects from the ith position on the search list in the corresponding ".RData" file (or "all.rda" file for image-loading packages, or "*.rdb/*.rdx" for lazyloading packages). There is less flexibility in the arguments than for the system equivalents. If you use the cd system in mvbutils, you will rarely need to call Save.pos directly; cd, move and FF will do it for you.

Usage

```
Save()
Save.pos( pos, path, ascii=FALSE)
```

Arguments

pos	string or numeric position on search path, or environment (e.g. ..mypack if "mypack" is a maintained-package).
path	directory or file to save into (see **Details**).
ascii	file type, as per save

Details

There is a safety provision in Save and Save.pos, which is normally invisible to the user, but can be helpful if there is a failure during the save process (for example, if the system shuts down unexpectedly). The workspace image is first saved under a name such as "n.RData" (the name will be adapted to avoid clashes if necessary). Then, if and only if the new image file has a different checksum to the old ".RData" file, the old file will be deleted and the new one will be renamed ".RData"; otherwise, the new file will be deleted. This also means that the ".RData" file will not be updated at all if there have been no changes, which may save time when synchronizing file systems or backing up.

Two categories of objects will not be saved by Save or Save.pos. The first category is anything named in options(dont.save); by default, this is ".packageName", ".SavedPlots", "last.warning", and ".Traceback", and you might want to add ".Last.value". The second category is anything which looks like a maintained package, i.e. an environment whose name starts with ".." and which has attributes "name", "path", and "task.tree". A warning will be given if such objects are found. [From bitter experience, this is to prevent accidents on re-loading after careless mistakes such as ..mypack$newfun <- something; what you *meant*, of course, is ..mypack$newfun <<- something. Note that the accident will not cause any bad effects during the current R session, because environments are not duplicated; anything you do to the "copy" will also affect the "real"

`..mypack`. However, a mismatch will occur if the environment is accidentally saved and re-loaded; hence the check in `Save`.]

`path` is normally inferred from the `path` attribute of the `pos` workspace. If no such attribute can be found (e.g. if the attached workspace was a list object), you will be prompted. If `path` is a directory, the file will be called ".RData" if that file already exists, or "R/all.rda" if that exists, or "R/*.rbd" for lazy loads if that exists; and if none of these exist already, then the file will be called ".RData" after all. If you specify `path`, it must be a complete directory path or file path (i.e. it will not be interpreted relative to a `path` attribute).

Compression: `mvbutils` uses the default compression options of `save`, unless you set `options()` "mvbutils.compress" and/or "mvbutils.compression_level" to appropriate values as per `?save`. The same applies to `mlazy` objects. Setting `options(mvbutils.compression_level=1)` can sometimes save quite a bit of time, at the cost of using more disk space. Set these options to NULL to return to the defaults.

History files: `Save` calls `savehistory()`. With package **mvbutils** from about version 2.5.6 on, `savehistory` and `loadhistory` will by default use the same file throughout each and every R session. That means everything works nicely for most users, and you really don't need to read the rest of this section unless you are unhappy with the default behaviour.

If you are unhappy, there are two things you might be unhappy about. First, `savehistory` and `loadhistory` are by default modified to always use the *current* value of the R_HISTFILE environment variable at the time they are called, whereas default R behaviour is to use the value when the session started, or ".Rhistory" in the current directory if none was set. I can't imagine why the default would be preferable, but if you do want to revert to it, then try to follow the instructions in `?mvbutils`, and email me if you get stuck. Second, the default for R_HISTFILE itself is set by `mvbutils` to be the file ".Rhistory" in the `.First.top.search` directory– normally the one you start R in. You can change that default by specifying R_HISTFILE yourself before loading `mvbutils`, in one of the many ways described by the R documentation on `?Startup` and `?Sys.getenv`.

Author(s)

Mark Bravington

See Also

`save`, `save.image`, `mtrace` in package **debug**, `mlazy`

Examples

```
## Not run:
Save() #
Save.pos( "package:mvbutils") # binary image of exported functions
Save.pos( 3, path="temp.Rdata") # path appended to attr( search()[3], "path")

## End(Not run)
```

search.for.regexpr *Find functions/objects/flatdoc-documentation containing a regexp.*

Description

Search one or more environments for objects that contain a regexp. Within each environment, check either (i) all functions, or (ii) the "doc" attributes of all functions, plus any character objects whose name ends in ".doc" or matches a specified regexp.

Usage

```
search.for.regexpr( pattern, where=1, lines=FALSE, doc=FALSE, code.only=FALSE, ...)
```

Arguments

pattern	the regexp
where	an environment, something that can be coerced to an environment (so the default corresponds to .GlobalEnv), or a list of environments or things that can be coerced to environments.
lines	if FALSE, return names of objects mentioning the regexp. If TRUE, return the actual lines containing the regexp.
doc	if FALSE, search function source code only. Otherwise, search the usual flatdoc places, i.e. "doc" attributes of functions, and certain character objects. If doc==TRUE, the name of those objects must end in ".doc"; otherwise, if doc is a string (length-1 character vector), then the names of the character object must grep that string; hence, doc="[.]doc$" is equivalent to doc=TRUE.
code.only	if FALSE, search only the deparsed version of "raw" code, so ignoring e.g. comments and "flatdoc" documentation
...	passed to grep– e.g. "fixed", "ignore.case".

Value

A list with one element per environment searched, containing either a vector of object names that mention the regexp, or a named list of objects & the actual lines mentioning the regexp.

See Also

flatdoc, find.docholder, find.documented

Examples

```
## Not run:
# On my own system's ROOT task (i.e. workspace--- see ?cd)
search.for.regexpr( 'author', doc=FALSE)
# $.GlobalEnv
# [1] "cleanup.refs"
```

```
# the code to function 'cleanup.refs' contains "author"
search.for.regexpr( 'author', doc=TRUE)
# $.GlobalEnv
# [1] "scrunge"
# 'scrunge' is a function with a character attribute that contains "author"
search.for.regexpr( 'author', doc='p')
#$.GlobalEnv
# [1] "scrunge" "p1"      "p2"
## 'scrunge' again, plus two character vectors whose names contain 'p'

## End(Not run)
```

`set.finalizer` *Obsolete but automatic finalization for persistent objects created in C.*

Description

[Almost certainly obsolete; `.Call` really is the way to go for newer code, complexity notwithstanding.]

Suppose you want to create persistent objects in C– i.e. objects that can be accessed from R by subsequent calls to C. The usual advice is that `.C` won't work safely because of uncertain disposal, and that you should use `.Call` and "externalptr" types instead. However, `.Call` etc is very complicated, and is much harder to use than `.C` in e.g. numerical settings. As an alternative, `set.finalizer` provides a safe way to ensure that your `.C`-created persistent object will tidy itself up when its R pointer is no longer required, just as you can with `externalptr` objects. There is no need for `on.exit` or other precautions.

Usage

```
# Always assign the result to a variable-- usually a temporary var inside a function...
# ... which R will destroy when the function ends. EG:
# keeper <- set.finalizer( handle, finalizer.name, PACKAGE=NULL)
set.finalizer( handle, finalizer.name, PACKAGE=NULL)
```

Arguments

`handle`	[integer vector]. Pointer to your object, of length 1 on 32-bit systems or 2 on 64-bit systems. Will have been returned by your object-creation function in C.
`finalizer.name`	Preferably a "native symbol" corresponding to a registered routine in a DLL; alternatively a string that names your `.C`-callable disposal routine. The routine must take exactly one argument, a 32-bit or 64-bit integer (the handle).
`PACKAGE`	[string] iff `finalizer.name` is `character`, this is a PACKAGE argument that specifies the DLL.

Details

You **must** assign the result to a variable, otherwise your object will be prematurely terminated!

`set.finalizer` provides a wrapper for R's own `reg.finalizer`, setting up a dummy "trigger" environment with a registered finalizer. The trigger is defined as an environment rather than the more obvious choice of an external pointer, because the latter would require me to get fancy with `.Call`. The role of `reg.finalizer` is to prime the trigger, so that when the trigger is subsequently garbage-collected, your specified `.C` function is called to do the finalization.

Note that finalization will only happen after *all copies* of `keeper` have been deleted. If you make a "temporary" copy in the global environment, remember to delete it! (Though presumably finalizers are de-registered if R is restarted and the keeper is reloaded, so there shouldn't be cross-session consequences.). Finalization won't necessarily happen immediately the last copy is deleted; you can call `gc()` to force it.

Value

A list with elements `handle` and `trigger`, the second being the environment that will trigger the call when discarded. The first is the original handle; it has storage mode integer so, as per **Examples**, you don't need to coerce it when subsequently passing it to `.C`.

See Also

.C, .Call, reg.finalizer

Examples

```
## Not run:
myfun <- function( ...) {
  ...0
  # Create object, return pointer, and ensure safe disposal
  keeper <- set.finalizer( .C( "create_thing", handle=integer(2), ...1)$handle,
      "dispose_of_thing")
  "cause" + "crash" # whoops, will cause crash: but finalizer will still be called
  # "dispose_of_thing" had better be the name of a DLL routine that takes a...
  # ... single integer argument, of length 1 or 2
  # Intention was to use the object. First param of DLL routine "use_thing" should
  # be pointer to thing.
  .C( "use_thing", keeper$handle, ...2)
}
myfun(...)

## End(Not run)
```

`setup.mcache` *Cacheing objects for lazy-load access*

Description

Manually setup existing reference objects– rarely used explicitly.

Usage

```
setup.mcache( envir, fpath, refs)
```

Arguments

envir	environment or position on the search path.
fpath	directory where "obj*.rda" files live.
refs	which objects to handle– all names in the mcache attribute of envir, by default

Details

Creates an active binding in envir for each element in refs. The active binding for an object myobj will be a function which keeps the real data in its own environment, reading and writing it as required. Writing a new value will give attr(envir, "mcache")["myobj"] a negative sign. This signals that the "obj*.rda" file needs updating, and the next Save (or move or cd) command will do so. [The "*" is the absolute value of attr(envir, "mcache")["myobj"].] One wrinkle is that the "real data" is initially a promise created by delayedAssign, which will fetch the data from disk the first time it is needed.

Author(s)

Mark Bravington

See Also

mlazy, makeActiveBinding, delayedAssign

---

sleuth *Generalized version of find*

Description

Looks for objects that regex-match pattern, in all attached workspaces (as per search()) and any maintained packages (see maintain.packages).

Usage

```
sleuth(pattern, ...)
```

Arguments

pattern	regex
...	other args to grep, e.g. perl=TRUE or ignore.case=TRUE

Value

A list of environments containing one or more matching objects, with the object names returned as a character vector within each list element.

See Also

search.for.regexpr

Examples

```
sleuth( '^rm')
# On my setup, that currently gives:
#$ROOT
#[1] "rmsrc"
#
#$`package:stats`
#[1] "rmultinom"
#
#$`package:base`
#[1] "rm"
#
#$mvbutils
#[1] "rm.pkg"
#
#$handy2
#[1] "rmultinom"
#
```

---

source.mvb *Read R code and data from a file or connection*

Description

source.mvb works like source(local=TRUE), except you can intersperse free-format data into your code. current.source returns the connection that's currently being read by source.mvb, so you can redirect input accordingly. To do this conveniently inside read.table, you can use from.here to read the next lines as data rather than R code.

Usage

```
source.mvb( con, envir=parent.frame(), max.n.expr=Inf,
  echo=getOption( 'verbose'), print.eval=echo,
  prompt.echo=getOption( 'prompt'), continue.echo=getOption( 'continue'))
current.source()
from.here( EOF=as.character(NA)) # Don't use it like this!
# Use "from.here" only inside "read.table", like so:
# read.table( file=from.here( EOF=), ...)
```

Arguments

con	a filename or connection
envir	an environment to evaluate the code in; by default, the environment of the caller of source

`max.n.expr`	finish after evaluating `max.n.expr` complete expressions, unless file ends first.
`EOF`	line which terminates data block; lines afterwards will again be treated as R statements.
`...`	other args to `read.table`
`echo, print.eval, prompt.echo, continue.echo`	as per `source`

Details

Calls to `source.mvb` can be nested, because the function maintains a stack of connections currently being read by `source.mvb`. The stack is stored in the list `source.list` in the `mvb.session.info` environment, on the search path. `current.source` returns the last (most recent) entry of `source.list`.

The sequence of operations differs from vanilla `source`, which parses the entire file and then executes each expression in turn; that's why it can't cope with interspersed data. Instead, `source.mvb` parses one statement, then executes it, then parses the next, then executes that, etc. Thus, if you include in your file a call to e.g.

```
text.line <- readLines( con=current.source(), n=1)
```

then the next line in the file will be read in to `text.line`, and execution will continue at the following line. `readLines.mvb` can be used to read text whose length is not known in advance, until a terminating string is encountered; lines after the terminator, if any, will again be evaluated as R expressions by `source.mvb`.

After `max.n.expr` statements (i.e. syntactically complete R expressions) have been executed, `source.mvb` will return.

If the connection was open when `source.mvb` is called, it is left open; otherwise, it is closed.

If you want to use `read.table` or `scan` etc. inside a `source.mvb` file, to read either a known number of lines or the rest of the file as data, you can use e.g. `read.table( current.source(), ...)`.

If you want to `read.table` to read an *unknown* number of lines until a terminator, you could explicitly use `readLines.mvb`, as shown in the demo "source.mvb.demo.R". However, the process is cumbersome because you have to explicitly open and close a `textConnection`. Instead, you can just use `read.table( from.here( EOF=...), ...)` with a non-default `EOF`, as in **Usage** and the same demo (but see **Note**). `from.here` *shouldn't* be used inside `scan`, however, because a temporary file will be left over.

`current.source()` can also be used inside a source file, to work out the source file's name. Of course, this will only work if the file is being handled by `source.mvb` rather than `source`.

If you type `source.list` at the R command prompt, you should always see an empty list, because all `source.mvb` calls should have finished. However, the source list can occasionally become corrupt, i.e. containing invalid connections (I have only had this happen when debugging `source.mvb` and quitting before the exit code can clean up). If so, you'll get an error message on typing `source.list` (?an R bug?). Normally this won't matter at all. If it bothers you, try `source.list <<- list()`.

Value

`source.mvb` returns the value of the last expression executed, but is mainly called for its side-effects of evaluating the code. `from.here` returns a connection, of class `c( "selfdeleting.file", "file", "connection")`; see **Details**. `current.source` returns a connection.

Limitations

Because source.mvb relies on pushBack, con=stdin() won't work.

Note

from.here creates a temporary file, which should be automatically deleted when read.table finishes (with or without an error). Technically, the connection returned by from.here is of class selfdeleting.file inheriting from file; this class has a specific close method, which unlinks the description field of the connection. This trick works inside read.table, which calls close explicitly, but not in scan or closeAllConnections, which ignore the selfdeleting.file class.

from.here() without an explicit terminator is equivalent to readLines(current.source()), and the latter avoids temporary files.

See Also

source, readLines.mvb, flatdoc, the demo in "source.mvb.demo.R"

Examples

```
# You wouldn"t normally do it like this:
tt <- tempfile()
cat( "data <- scan( current.source(), what=list( x=0, y=0))",
"27 3",
"35 5",
file=tt, sep="\n")
source.mvb( tt)
unlink( tt)
data # list( x=c( 27, 35), y=c(3, 5))
# "current.source", useful for hacking:
tt <- tempfile()
cat( "cat( \"This code is being read from file\",",
"summary( current.source())$description)", file=tt)
source.mvb( tt)
cat( "\nTo prove the point:\n")
cat( scan( tt, what="", sep="\n"), sep="\n")
unlink( tt)
```

---

strip.missing *Exclude "missing" objects*

Description

To be called inside a function, with a character vector of object names in that function's frame. strip.missing will return all names except those corresponding to formal arguments which were not set in the original call and which lack defaults. The output can safely be passed to get.

Usage

```
strip.missing( obs)
```

Arguments

obs	character vector of object names, often from ls(all=TRUE)

Details

Formal arguments that were not passed explicitly, but which **do** have defaults, will **not** be treated as missing; instead, they will be set equal to their evaluated defaults. This could cause problems if the defaults aren't meant to be evaluated.

Author(s)

Mark Bravington

See Also

returnList

Examples

```
funco <- function( first, second, third) {
  a <- 9
  return( do.call("returnList", lapply( strip.missing( ls()), as.name)))
}
funco( 1) # list( a=9, first=1)
funco( second=2) # list( a=9, second=2)
funco( ,,3) # list( a=9, third=3)
funco2 <- function( first=999) {
  a <- 9
  return( do.call("returnList", lapply( strip.missing( ls()), as.name)))
}
funco2() # list( a=9, first=999) even tho' "first" was not set
```

---

task.home *Organizing R workspaces*

Description

Returns file path to current task, or to a file in that task.

Usage

```
# Often: task.home()
task.home(fname)
```

Arguments

fname	file name, a character(1)

Details

Without any arguments, `task.home` returns the path of the current task. With a filename argument, the filename is interpreted as relative to the current task, and its full (non-relative) path is returned.

`task.home` is almost obsolete in R, since the working directory tracks the current task. It is more important in the S+ version of `mvbutils`.

Author(s)

Mark Bravington

See Also

`cd`, `getwd`, `file.path`

Examples

```
## Not run:
task.home( "myfile.c") # probably the same as file.path( getwd(), "myfile.c")
task.home() # probably the same as getwd()

## End(Not run)
```

---

unpackage *Convert existing source package into task package*

Description

Converts an existing source package into a task package. A subdirectory with the package name will be created under the current working directory, and will be populated with a ".RData" file and various other files/directories from the source package. All Rd files will be turned into flat-format help in the ".RData", either attached to functions or as stand-alone "*.doc" text objects, as per `help2flatdoc`. The subdirectory will also be made into a *task*, i.e. it will be added to the "tasks" vector in the current workspace that `cd` uses to keep track of the task hierarchy.

Usage

```
unpackage(spath, force = FALSE)
```

Arguments

`spath`	where to find the source package
`force`	if TRUE, overwrite any previous contents of task package without prompting.

Details

The NAMESPACE file won't be copied; instead, it will be auto-generated by `pre.install`. Therefore, some features of the original NAMESPACE may be lost. You can either copy the NAMESPACE manually (in which case, you'll need to maintain it by hand), or write a "pre.install.hook.MYPACK" function.

The DESCRIPTION file becomes a character vector called e.g. `<mypack>.DESCRIPTION`, of class "cat" (see `as.cat`).

Functions in the ".RData" may be saved with extra attributes, in particular "doc" (deduced from a dot-Rd file) but perhaps other things too that they acquire after the package code is `sourced`. Attributes that are character vector will acquire class "docattr", so that they won't be fully displayed during default printing of the function; to see them, use e.g. `as.cat( attr( myfun, "myatt"))` or `unclass( attr( myfun, "myatt"))` or, if you are using the **atease** package, just `as.cat( myfun@myatt)`. Editing the function with `fixr` will display the character attributes in full.

Any environment objects found in the package's environment (its namespace environment) will be dropped from the ".RData" file, with a warning; this is to avoid dramas on reloading.

See Also

`pre.install`, `mvbutils.packaging.tools`

`warn.and.subset` *Extract subset and warn about omitted cases*

Description

Extract row-subset of a `data.frame` according to a condition. If any cases (rows) are omitted, they are listed with a warning. Rows where the condition gives NA are omitted.

Usage

```
# This is the obligatory format, and is not very useful; look at EXAMPLES instead
warn.and.subset(x, cond,
    mess.head=deparse( substitute( x), width.cutoff=20, control=NULL, nlines=1),
   mess.cond=deparse( substitute( cond), width.cutoff=40, control=NULL, nlines=1),
    row.info=rownames( x), sub=TRUE)
```

Arguments

`x`	data.frame
`cond`	expression to evaluate in the context of `data.frame`. If `sub=TRUE` (the default), this will be substituted. If `sub=FALSE`, you can use a pre-assigned expression; in that case, you had better set `mess.cond` manually.
`mess.head`	description of data.frame (e.g. its name) for use in a warning.
`mess.cond`	description of the desired condition for use in a warning.
`row.info`	character vector that will describe rows; omitted elements appear in the warning

sub — should cond be treated as a literal expression to be evaluated, or as a pre-computed logical index? # ...: just there to keep RCMD CHECK happy– for heaven's sake...

Value

The subsetted data.frame.

See Also

`%where.warn%` which is a less-flexible way of doing the same thing

Examples

```
df <- data.frame( a=1:3, b=letters[1:3])
df1 <- warn.and.subset( df, a %% 2 == 1, 'Boring example data.frame', 'even-valued "a"')
condo <- quote( a %% 2 == 1)
df2 <- warn.and.subset( df, condo, 'Same boring data.frame', deparse( condo), sub=FALSE)
```

Index

∗ **IO**
readLines.mvb, 108
∗ **data**
mlazy, 63
setup.mcache, 114
∗ **debugging**
Save, 110
∗ **documentation**
dochelp, 23
find.documented, 27
flatdoc, 36
get.backup, 43
∗ **file**
Save, 110
∗ **misc**
changed.funs, 13
check.patch.versions, 13
ditto.list, 14
do.on, 16
dont.lockBindings, 25
fast.read.fwf, 27
foodweb, 38
generic.dll.loader, 41
help, 46
install.pkg, 50
library.dynam.reg, 53
lsize, 56
maintain.packages, 57
make_dull, 60
max_pkg_ver, 61
mcut, 62
multirep, 71
mvbutils-package, 3
mvbutils.operators, 74
mvbutils.packaging.tools, 76
mvbutils.utils, 83
NEG, 90
noice, 91
print, 100
rbdf, 104
rm.pkg, 109
search.for.regexpr, 112
set.finalizer, 113
sleuth, 115
source.mvb, 116
unpackage, 120
warn.and.subset, 121
∗ **programming**
do.in.envir, 15
doc2Rd, 17
dont.lock.me, 24
extract.named, 26
find.documented, 27
fixr, 30
flatdoc, 36
get.backup, 43
hack, 45
help2flatdoc, 49
local.on.exit, 54
local.return, 55
lsize, 56
make.NAMESPACE, 59
mlazy, 63
mlocal, 67
mvb.sys.parent, 72
my.index, 89
pre.install, 92
setup.mcache, 114
strip.missing, 118
∗ **utilities**
cd, 6
cdfind, 10
cdprompt, 12
do.in.envir, 15
find.documented, 27
fix.order, 29
fixr, 30
get.backup, 43

make.NAMESPACE, 59
move, 69
my.index, 89
pre.install, 92
task.home, 119
? (help), 46
%!in% (mvbutils.operators), 74
%**% (mvbutils.operators), 74
%<-% (mvbutils.operators), 74
%SUCH.THAT% (mvbutils.operators), 74
%&% (mvbutils.operators), 74
%downto% (mvbutils.operators), 74
%except% (mvbutils.operators), 74
%grepling% (mvbutils.operators), 74
%in.range% (mvbutils.operators), 74
%is.a% (mvbutils.operators), 74
%is.an% (mvbutils.operators), 74
%is.not.a% (mvbutils.operators), 74
%is.not.an% (mvbutils.operators), 74
%matching% (mvbutils.operators), 74
%not.in.range% (mvbutils.operators), 74
%not.in% (mvbutils.operators), 74
%perling% (mvbutils.operators), 74
%such.that% (mvbutils.operators), 74
%that.are.in% (mvbutils.operators), 74
%that.dont.match% (mvbutils.operators), 74
%that.match% (mvbutils.operators), 74
%upto% (mvbutils.operators), 74
%where.warn% (mvbutils.operators), 74
%where% (mvbutils.operators), 74
%without.name% (mvbutils.operators), 74

as.cat, *91*, *101*, *121*
as.cat (mvbutils.utils), 83
assign.to.base (hack), 45
attach.mlazy (mlazy), 63
autoedit (fixr), 30

build.pkg, *14*, *78–82*, *96*
build.pkg (install.pkg), 50
build.pkg.binary, *14*, *78*, *81*

callees.of (foodweb), 38
callers.of (foodweb), 38
cbind, *104*, *105*
cd, *3–6*, 6, *10–12*, *32*, *35*, *44*, *57*, *58*, *63–66*, *69*, *71*, *77*, *94*, *97*, *99*, *110*, *115*, *120*
cd.change.all.paths, *8*, *10*
cd.change.all.paths (cdfind), 10
cd.write.mvb.tasks, *10*
cd.write.mvb.tasks (cdfind), 10
cdfind, *10*, 10
cditerate, *10*
cditerate (cdfind), 10
cdprompt, *10*, 12
cdregexpr (cdfind), 10
cdtree, *10*
cdtree (cdfind), 10
changed.funs, 13
check.patch.versions, 13
check.pkg, *81*
check.pkg (install.pkg), 50
clip (mvbutils.utils), 83
cq (mvbutils.utils), 83
create.backups (get.backup), 43
create.wrappers.for.dll (generic.dll.loader), 41
cull.old.builds, *82*
cull.old.builds (install.pkg), 50
current.source, *108*
current.source (source.mvb), 116
cut, *62*

data, *97*
demlazy (mlazy), 63
deparse.names.parsably (mvbutils.utils), 83
ditto.list, 14
do.in.envir, *6*, 15, *55*, *69*
do.on, 16
doc2Rd, 17, *24*, *28*, *37*, *49*, *50*, *77–79*, *97*, *99*
dochelp, *4*, *6*, *18*, *20*, 23, *28*, *37*, *46*, *49*
docotest (doc2Rd), 17
dont.lock.me, 24
dont.lockBindings, 25

empty.data.frame, *104*, *106*
empty.data.frame (mvbutils.utils), 83
env.name.string (mvbutils.utils), 83
everyth (mvbutils.utils), 83
expanded.call (mvbutils.utils), 83
extract.named, *3*, 26

fast.read.fwf, 27
FF, *37*, *43*, *110*
FF (fixr), 30
find.docholder, *112*

find.docholder *(find.documented)*, 27
find.documented, *24*, 27, *112*
find.funs *(mvbutils.utils)*, 83
find.lurking.envs, *81*
find.lurking.envs *(mvbutils.utils)*, 83
fix.order, 29, *35*, *44*
fixr, *3–8*, *10*, *17*, *29*, 30, *36*, *37*, *43*, *44*, *57–59*, *77*, *78*, *80*, *100*, *121*
fixtext, *58*, *79*
fixtext *(fixr)*, 30
flatdoc, *6*, *22*, *24*, *28*, *33*, 36, *49*, *50*, *60*, *78*, *95*, *97*, *108*, *112*, *118*
foodweb, *3*, *6*, *11*, 38
FOR, *3*
FOR *(do.on)*, 16
from.here *(source.mvb)*, 116

generic.dll.loader, 41, *80*
get.backup, *35*, 43

hack, 45
help, *3–5*, *18*, *23*, *33*, *36*, 46, *50*
help2flatdoc, *17*, *18*, *22*, 49, *102*, *120*

index *(mvbutils.utils)*, 83
install.pkg, 50, *77*, *79–81*, *93*, *96*
integ *(mvbutils.utils)*, 83
is.dir *(mvbutils.utils)*, 83
isF *(mvbutils.utils)*, 83
isT *(mvbutils.utils)*, 83

ldyn.tester *(generic.dll.loader)*, 41
ldyn.unload *(generic.dll.loader)*, 41
legal.filename *(mvbutils.utils)*, 83
library.dynam.reg, 53
load.refdb, *66*
local, *33*
local.on.exit, 54, *68*, *69*
local.return, *55*, 55, *68*, *69*
localfuncs, *69*
lsall *(mvbutils.utils)*, 83
lsize, 56, *66*, *67*

maintain.packages, *6*, *24*, *31*, 57, *70*, *77*, *78*, *80*, *93*, *99*, *109*, *115*
make.arguments.section, *37*
make.dull *(make_dull)*, 60
make.NAMESPACE, 59, *95*
make.usage.section, *37*, *98*
make_dull, 60
makeActiveBinding, *66*
masked *(mvbutils.utils)*, 83
masking *(mvbutils.utils)*, 83
massrep *(multirep)*, 71
max_pkg_ver, 61
mcachees *(mlazy)*, 63
mcut, *3*, 62
mintcut *(mcut)*, 62
mkdir *(mvbutils.utils)*, 83
mlazy, *3*, *5*, *6*, *9*, *10*, *56*, 63, *87*, *95*, *97*, *110*, *111*, *115*
mlocal, *3*, *6*, *15*, *16*, *33*, *45*, *54–56*, 67, *87*
most.recent *(mvbutils.utils)*, 83
move, *5*, *10*, *32*, *35*, *43*, *44*, *58*, *65*, 69, *80*, *110*, *115*
mtidy *(mlazy)*, 63
multinsert *(multirep)*, 71
multirep, *3*, 71
mvb.eval.parent *(mvb.sys.parent)*, 72
mvb.match.call *(mvb.sys.parent)*, 72
mvb.nargs *(mvb.sys.parent)*, 72
mvb.parent.frame *(mvb.sys.parent)*, 72
mvb.sys.call *(mvb.sys.parent)*, 72
mvb.sys.function *(mvb.sys.parent)*, 72
mvb.sys.nframe *(mvb.sys.parent)*, 72
mvb.sys.parent, 72
mvbutils *(mvbutils-package)*, 3
mvbutils-package, 3
mvbutils.operators, *3*, *6*, 74
mvbutils.packaging.tools, *3*, *4*, *6*, *8*, *10*, *13*, *33*, *36*, *50*, *57*, *59*, 76, *92*, *94*, *95*, *97–99*, *121*
mvbutils.utils, *3*, *6*, 83
my.all.equal *(mvbutils.utils)*, 83
my.index, 89

named, *16*
named *(mvbutils.utils)*, 83
NEG, 90
noice, 91
nscat *(mvbutils.utils)*, 83
nscatn *(mvbutils.utils)*, 83

option.or.default *(mvbutils.utils)*, 83

par, *39*
patch.install, *13*, *14*, *17*, *52*, *58*, *77*, *79*, *80*
patch.install *(pre.install)*, 92

patch.installed, *59*
patch.installed *(*pre.install*)*, 92
plot.cdtree *(*cdfind*)*, 10
plot.foodweb *(*foodweb*)*, 38
pos, *4*, *10*, *27*, *111*
pos *(*mvbutils.utils*)*, 83
pre.install, *13*, *17*, *19, 20*, *22*, *50*, *52*, *58–60*, *77*, *79–83*, 92, *121*
print, *32*, *84, 85*, 100
print.cat, *21*, *32*
print.function, *4, 5*, *33*
put.in.session *(*mvbutils.utils*)*, 83

rbdf, 104
rbind, *4*, *84, 85*, *103*
rbind *(*rbdf*)*, 104
rbind.data.frame, *4, 5*
read.bkind *(*get.backup*)*, 43
read.table, *27*
readLines.mvb, 108, *117, 118*
readr *(*fixr*)*, 30
remove.from.package *(*rm.pkg*)*, 109
returnList, *55*, *119*
returnList *(*mvbutils.utils*)*, 83
rm.pkg, *80*, 109

safe.rbind *(*mvbutils.utils*)*, 83
sapply, *16*
Save, *5*, *9*, *32*, *44*, *58*, *63–66*, *70*, 110, *115*
save, *110, 111*
Save.pos, *44*, *70*, *77*
scatn, *85*, *87*
scatn *(*mvbutils.utils*)*, 83
search.for.regexpr, *3*, 112, *116*
set.finalizer, *42*, 113
set.rcmd.vars *(*install.pkg*)*, 50
setup.mcache, 114
sleuth, 115
source.mvb, *4*, *6*, *37*, *108*, 116
spkg, *81*
spkg *(*pre.install*)*, 92
startDynamicHelp, *47*
strip.missing, *3*, 118
strwrap, *23*

task.home, *9, 10*, *64*, 119
to.regexpr *(*mvbutils.utils*)*, 83

unmaintain.package *(*maintain.packages*)*, *57*
unpackage, *50*, *57*, *59*, 120

warn.and.subset, 121
write.NAMESPACE *(*make.NAMESPACE*)*, 59
write.sourceable.function, *22*, *37*

yes.no *(*mvbutils.utils*)*, 83