

Package ‘mvbutils’

February 14, 2012

Version 2.5.101

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

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

Depends R (>= 2.13), utils, tools, stats

Date 30/10/2011

Title Workspace organization, code and documentation editing, package prep and editing, etc.

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)

ChangeLog inst/changes.txt

Repository CRAN

Date/Publication 2011-11-02 17:26:04

R topics documented:

mvbutils-package	2
cd	6
cdfind	10
cdprompt	12
changed.funs	13
do.in.envir	13
doc2Rd	15
dochelp	20
dont.lock.me	21
dont.lockBindings	22
extract.named	23
fast.read.fwf	24

find.documented	24
fix.order	26
fixr	27
flatdoc	31
foodweb	33
get.backup	35
hack	38
help2flatdoc	39
Hours	40
install.pkg	40
local.on.exit	42
local.return	43
lsize	44
make.NAMESPACE	45
mcut	46
mlazy	47
mlocal	51
move	53
multirep	55
mvb.sys.parent	56
mvbutils.operators	57
mvbutils.packaging.tools	59
mvbutils.utils	64
my.index	70
my.package.path	72
pre.install	73
readLines.mvb	80
rm.pkg	81
Save	82
search.for.regexpr	84
setup.mcache	85
source.mvb	85
strip.missing	88
task.home	89
unpackage	90
warn.and.subset	91

Index**92**

Description

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

- Hierarchical organization of projects (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 object 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 that generate general-purpose objects can also be maintained this way. Function documentation can be stored as plain text after the function definition, and will be found by `help`. 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](#)), better handling of POSIXt especially in matrices & data.frames, numerous lower-level utility functions and operators ([mvbutils.utils](#), [mvbutils.operators](#), [extract.named](#), [mcut](#), [search.for.regexpr](#), [strip.missing](#), [Hours](#)).

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. `help`
- `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 above

Redefined functions

On loading, the present version of package **mvbutils** compulsorily overwrites a few system functions: `library`, `rbind.data.frame`, `lockEnvironment`, `loadNamespace`. It also creates methods `rbind.POSIXct`, `cbind.POSIXct`, and `length<- .POSIXct`. By default, it also overwrites `help`, `savehistory`, `loadhistory`, `save.image`, `difftime`, `+ .POSIXt`, `- .POSIXt`, `head.matrix`, and `tail.matrix`. (The original version of routine `xxx` can always be obtained via `base.xxx` if you really need it.) The mods are undone when you unload `mvbutils`. The mods should have [almost] no side-effects, and/but I hope to be able to avoid them altogether in future versions of R (hasn't happened in 5 years, though). Unloading `mvbutils` undoes the changes. Briefly:

- `library` is modified so that its default `pos` argument becomes a call to `lib.pos()`. This means that packages get attached just below `ROOT` rather than always in position 2 (needed by `cd`).
- `lockEnvironment` is modified to allow live-editing of 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` is modified to work better (IMO) when the first `data.frame` has zero rows, to cope with a code-breaking change in R's behaviour several versions ago. Specifically, the modified version does *not* drop zero-row `data.frames`, and their column attributes are taken account of when `rbind.data.frame` ing to the other args. This is useful when repeatedly adding rows to an initially-empty `data.frame`. To see the difference, experiment with `rbind(data.frame(x=1, y=factor("a")), data.frame(x=2, y=factor("b"))[-1,])$y` vs `base.rbind.data.frame(...)` with the same arguments. `mvbutils` and `debug` rely on the non-default behaviour, so the overwriting is not optional.
- `rbind.data.frame` is also modified so that dimensioned elements (i.e. matrices & arrays within `data.frames`) do not lose extra attributes; hence you can `rbind` two `data.frames` that both have `POSIXct`-matrix elements without turning them into raw seconds and losing timezones.

Optional but recommended replacements are as follows:

- `help` is modified so that, if system `help` can't find help for a function (but not a method, dataset, or package), it will look for a `doc` attribute of the function to display in a pager using `dochehelp`.
- `loadhistory` and `savehistory` are modified so that they use the "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. 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.
- `difftime`, `+.POSIXt`, and `-.POSIXt` are modified to behave more consistently and forgivingly. Results won't break code that doesn't make invalid assumptions. [This should probably be done in a different package. But no-one has complained yet, so the mods will stay until/unless enough people do...]
- `print.POSIXct`, `format.POSIXct`, `as.data.frame.POSIXt`, and the new methods `rbind.POSIXct` & `cbind.POSIXct` & `length<-.POSIXct`, are modified/created to honour any matrix/array shape of `POSIXct` objects. Without this mod, R forces you to either hide the matrix shape if you want to see the POSIXity, or to discard POSIXity if you want to see the matrixity. I'm not sure that a `POSIXlt` matrix makes sense, so I haven't provided comparable mods for `POSIXlt` yet.
- `head.default` and `tail.default` are modified to call `head/matrix` if the argument is a matrix. Although there are already `head/matrix` methods, by default they won't be invoked for 2D objects that have a non-default S3 class, such as `POSIXct` objects. This mod fixes the problem.
- `update.default` will check for a `attr(x, "call")` if there is no component `x$call`.

If you are certain that you don't want the optional replacements, set `options(mvbutils.replacements=FALSE)` before loading `mvbutils` (though `rbind.POSIXct`, `cbind.POSIXct`, `length<-.POSIXct` are currently compulsory, so will be added regardless). However, this will prevent `cd`, `fixr`, and the flat-documentation help from working properly. You can also set the "mvbutils.replacements" option to a character vector comprising some or all of the above names.

After `mvbutils` has loaded, you can undo the modification of an individual function called `xxx` with `assign.to.base("xxx", base.xxx)`. Unloading `mvbutils` will undo all the changes.

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 5 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](#), 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 <code>.First.task</code> code be executed on attachment? Yes, unless there's a bug in it.
execute.Last	should the <code>.Last.task</code> 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 sourcing 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` always loads packages below `ROOT`, the `.First.lib` code in `mvbutils` makes a minor hack to `library`, setting the default `pos` argument to call `lib.pos()` which will locate the appropriate search position.

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 lacking 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, ...) # plot.cdtree; normally plot( cdtree(<<args>>))

```

Arguments

pattern	regexp to be checked against object names.
regexp	regexp 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 <code>par(cex)</code>
old.path	regexp 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 <code>cditerate</code>), or to foodweb (for <code>plot.cdtree</code>)
x	result of a call to <code>cdtree</code> , 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 hierarchy (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	<i>Show functions and callees in environment 'egood' that have changed or disappeared in environment 'ebad'.</i>
--------------	--

Description

Useful eg when you have been modifying a package, and have bugged 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 <code>foodweb(egood)</code> , but this will be called automatically if not.

Value

Character vector with the names of changed/lost functions.

do.in.envir	<i>Modify a function's scope</i>
-------------	----------------------------------

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)
```

doc2Rd

Converts plain-text documentation to Rd format

Description

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. `help(glm, help_type="text")` and try making one yourself. Don't bother with indentation, except in item lists as per MORE.DETAILS below. See `fixr` and its `new.doc` argument for how to set up an empty template.

Usage

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

Arguments

<code>text</code>	(character or function) character vector of documentation, or a function with a <code>doc</code> attribute that is a c.v. of d..
<code>file</code>	(string or connection) if non-NULL, write the output to this file
<code>append</code>	(logical) only applies if <code>!is.null(file)</code> ; should output be appended rather than overwriting?
<code>warnings.on</code>	(logical) ?display warnings about apparently informal documentation?
<code>Rd.version</code>	(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.
<code>def.valids</code>	(character) objects or helpfiles for which links should be generated automatically. When <code>doc2Rd</code> is being called from <code>pre.install</code> , 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).
<code>check.legality</code>	if TRUE and <code>Rd.version</code> is 2 or more, then the output Rd will be run thru <code>parse_Rd</code> and a <code>try-error</code> will be returned if that fails; normal return otherwise. Not applicable if <code>Rd.version</code> is 1.

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 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. 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(..., pager=T)` (pre-2.10) or `help(..., help_type="text")` (post-2.10).

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 (i.e. not HTML help) 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** below.
- "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). 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; see **Special fields** below.

Subsections: 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 `doche1p`, though, so can be used to hide an unhelpful USAGE, say, or to hide an `#ifdef windows`.

Triple dots are converted to `\dots`, regardless of whether they're in code or normal text. A solitary capital-R is converted to `\R`.

Any reasonable `"*b*old"` or `"_emphatic stuff_"` constructions (no quotes, just the asterisks) will go into `\bold{}` and `\emph{}` constructs respectively, to give **bold** 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 a 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, 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. Cross-refs to (sub)sections will be picked up if of the form "see ANOTHER.SECTION" or "see also ANOTHER.SECTION" or "ANOTHER.SECTION (qv)". Cross-refs to subsections should omit the initial full stops.

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, 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.

In KEYWORDS, 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:** 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 can be documented just like any other function, except for one small detail: in the USAGE section, the generic name instead of your method name should be used. IE if you are documenting a method `print.cat`, the USAGE section should contain a call to `print(x,...)` rather than `print.cat(x,...)`. doc2Rd will then detect your intent and produce correct Rd format. If you describe several different S3 methods for the same generic in the same piece of documentation (and even if you don't), then it may help the user if you put a comment with the method name after each reference in USAGE, e.g. `print(x,...) # print.cat`, especially if the optional arguments are different. Doing that also gives `pre.install` a better chance of correctly sorting out the documentation.

Confusion will arise with a function that looks like an S3 method, but aren'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 this.

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.

Author(s)

Mark Bravington

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: "RCMD Rd2dvi -pdf XXX.Rd" and "RCMD Rdconv -t=html XXX.Rd" and/or "-t=txt" (though `patch.installed` will allow to check the HTML version immediately).

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'))
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
```

dochelp

Documentation

Description

dochelp(topic) will be invoked by the replacement help if conventional help fails to find documentation for topic topic. If topic has a doc attribute, the latter will be formatted and displayed by file.show. dochelp is not usually called directly.

Usage

```
# Not usually called directly
# If it is, then normal usage is: dochelp( topic)
dochelp( topic, doc)
```

Arguments

topic	(character) name of the object to look for help on.
doc	(character or list)– normally not set, but deduced by default from topic; see Details .

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.

The file 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](#)).

[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.

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",
  docheat( "myfun" )
  help( "myfun" ) # calls docheat
```

dont.lock.me	<i>Prevent sealing of a namespace, to facilitate package maintenance.</i>
--------------	---

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

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

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	<i>Create variables from corresponding named list elements</i>
---------------	--

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	<i>Read in fixed-width files quickly</i>
---------------	--

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
```

Arguments

<code>file</code>	character
<code>width</code>	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.
<code>col.names</code>	names for the columns that are NOT skipped
<code>colClasses</code>	can be used to control type conversion; see read.table . It is an optional vector whose names must be part of <code>col.names</code> . There is one extension of the <code>read.table</code> rules: a <code>colClass</code> string starting <code>POSIXct.</code> will trigger automatic conversion to <code>POSIXct</code> , using the rest of the string as the format specifier. See also <code>tz</code> .
<code>na.strings</code>	are there any strings (other than NA) which should convert to NAs?
<code>tz</code>	used in auto-conversion to <code>POSIXct</code> when <code>colClass</code> is set
<code>...</code>	ignored; it's here so that this function can be called just like <code>read.fwf</code>

Value

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

find.documented	<i>Support for flat-format documentation</i>
-----------------	--

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.dochoolder` says where the documentation for one or more functions is actually stored. Both `find.documented` and `find.dochoolder` 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.docheolder( what, pos=find( funs[1]))
```

Arguments

pos search path position(s), numeric or character. In `find.documented`, any length. In `find.docheolder`, only `pos[1]` will be used; it defaults to where the first element of `funs` 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`; "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.docheolder`
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](#), [dochehelp](#)

fix.order	<i>Shows functions sorted by date of edit</i>
-----------	---

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
fixr.order() # functions in .GlobalEnv
fixr.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 scriptlet for making a single object of any type) 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)`, or manually by calling `FF()`. There is an optional automatic text backup facility. `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. `fixrtext` is a shorthand form for forcing creation of a text object rather than the default of a function. For non-function and non-text objects, you can edit a scriptlet that creates the object and is stored as its source attribute. The object is updated whenever the scriptlet is changed, by running the scriptlet; see separate section below.

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)
# fixrtext really has exact same args as fixr, but technically its args are:
fixrtext( 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 general object. You can also write `mypack$myfun`, or `mypack::myfun`, or `mypack:::myfun`, or `..mypack$myfun`, to simultaneously set the `pkg` argument. 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.

<code>new.doc</code>	(logical) if TRUE, add skeleton plain-text R-style documentation, as per <code>add.flatdoc.to</code> . Also use this to create an empty scriptlet for a general (non-function, non-text) object.
<code>new</code>	(logical, seldom used) if TRUE, edit a blank function template rather than any existing copy in the search path. New edit will go into <code>.GlobalEnv</code> unless argument <code>pkg</code> is set.
<code>install</code>	(logical, rarely used) logical indicating whether to go through the process of asking you about your editor
<code>what</code>	(logical, rarely used) if no pre-existing <code>x</code> , then <code>fixr</code> creates an empty function template by default. Set <code>what=""</code> to create an empty character vector instead—or just use <code>fixtext</code> .
<code>fixing</code>	(logical, rarely used) FALSE for read-only (i.e. just opening editor to examine the object)
<code>pkg</code>	(string or environment) if non-NULL, then specifies in which package a specific maintained package (see maintain.packages) <code>x</code> should be looked for.
<code>do</code>	(logical) TRUE => automatically update objects from altered files; FALSE => don't.
<code>...</code>	other arguments, except what in <code>fixtext</code> , and <code>fixing</code> in <code>readr</code> , are passed to <code>fixr</code> .

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 [moved](#) it.

If something goes wrong during an automatic call to `FF` (very unusual), the automatic-call feature will stop working. To get it back 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.

`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, it's wrapped in a `structure(...)` construct. 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`).

If the function was being `mtraced` (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.

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 functions by edit date; see `?fix.order`. Backup currently only works for functions and character objects.

Scriptlets for general objects

`fixr` can edit an object of any type, within reason; more precisely, it can edit a "scriptlet" of R code that generates the object. The scriptlet is evaluated as soon as FF detects a changed file, and the result is assigned to the object. Only the first complete expression in a scriptlet is evaluated; use braces to group multiple expressions. The scriptlet should not include the assignment to the final object; e.g., the scriptlet for `myobj` should be something like `1:10`, not `myobj <- 1:10`. Evaluation takes place in a temporary environment inheriting from `.GlobalEnv`, so side-effect assignments during the scriptlet are discarded. The scriptlet itself is stored as the "source" attribute of the object, of class `cat`.

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`.

Note that the second case is not meant to be evaluated immediately— I often just want to save a call, not the result of the call. The solution is to wrap the statement in a call to `quote()` or `expression()`.

```
{ # Brace needed because several expressions are involved
  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)
  })
  raw.data
}
```

To edit a new general object `myobj`, call `fixr(myobj, what=list())`; it doesn't matter whether `myobj` will actually be a list. If you forget the `what` argument, you can just replace the function template with your scriptlet; the `fixr` paradigm for general objects is the same as for functions. However, omitting `what=list()` can lead to problems if your scriptlet fails to parse.

If you want to use `fixr` to edit a general object that wasn't created with `fixr`, just set `new.doc=TRUE`; you shouldn't need the `what` argument.

If you're calling `fixr` on an object that already has a scriptlet, the scriptlet will appear and you edit that.

You can use the **debug** package on scriptlets. To turn debugging on/off, use `mtrace(eval.scriptlet)` and `mtrace(eval.scriptlet, F)`. Currently, there is no way to select particular scriptlets; either all are debugged during updating, or none are.

The backup files for general objects show the scriptlet, not the result.

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 the only way of using 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 nothing about). Input welcome (but none has been forthcoming for several years!).

See Also

.First, [edit](#), [cd](#), [get.backup](#), [fix.order](#), [move](#)

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 <code>readlines.mvb</code>
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

```
### Don't run
### Put everything before the next comment into a text file <<your filename>>
#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 everything before the next comment into a text file
#myfun <- structure( function( atname) {
#  attr( sys.function(), atname)
#}
#, 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 don't 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)
## S3 method for class 'foodweb'
plot(x, textcolor, boxcolor, xblank, border, textargs = list(), use.centres = TRUE, color.lines = TRUE,
callers.of( funs, fw=foodweb( plotting=FALSE))
callees.of( funs, fw=foodweb( plotting=FALSE))

```

Arguments

<code>funs</code>	character vector OR (in foodweb only) the result of a previous foodweb call
<code>where</code>	position(s) on search path, or an environment, or a list of environments
<code>charlim</code>	controls maximum number of characters per horizontal line of plot
<code>prune</code>	character vector. If omitted, all funs will be shown; otherwise, only ancestors and descendants of functions in <code>prune</code> will be shown. Augments funs if required.
<code>rprune</code>	regex version of <code>prune</code> ; <code>prune <- funs %matching% rprune</code> . Does NOT augment funs. Overrides <code>prune</code> if set.
<code>ancestors</code>	show ancestors of <code>prune</code> functions?
<code>descendents</code>	show descendents of <code>prune</code> functions?
<code>plotting</code>	graphical display?
<code>plotmath</code>	leave alone
<code>generics</code>	calls TO functions in generics won't be shown
<code>lwd</code>	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)

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 `funcs`, `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.

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
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="package:mvbutils", rprune="name") # does work
foodweb( where=asNamespace( "mvbutils")) # secret stuff
fw <- foodweb( where="package:mvbutils")
fw$funmat # a big matrix
callers.of( "mlocal", fw)
callees.of( find.funs() %matching% "name", fw)

```

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. <code>..mypack</code> 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 <code>1+length(object)</code> . 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 `moved`) 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-unbacked 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

<code>get.backup</code>	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 cur-
                rently 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	<i>Modify standard R functions, including tweaking their default arguments</i>
------	--

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

<code>fun</code>	a function (not a character string)
<code>...</code>	pairlist of arguments and new default values, e.g. <code>arg1=1+2</code> . Things on RHS of equal signs will not be evaluated.
<code>x</code>	function name (a character string)
<code>what</code>	function to replace <code>x</code> , defaulting to <code>"replacement."</code> <code>%%&& x</code>
<code>where</code>	where to find the replacement function, defaulting to usual search path
<code>in.imports</code>	usually TRUE, if this is being called from an <code>.onLoad</code> 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 .
<code>override.env</code>	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)
```

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.

Usage

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

Arguments

fun.name	function name (a character string)
pkg	name of package
text	plain-text help 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.

Aliases are deduced from function calls in the USAGE section.

(Link-triggering phrases aren't explicitly created— could look thru lists of linkable things I guess.)

See Also

[doc2Rd](#)

Examples

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

 Hours

Sorting out times and time difference

Description

These functions enforce seconds-based calculations of time differences. Hours, Minutes, and Seconds all return the obvious number of seconds, together with the attribute `unit="seconds"`. Further, when you load `mvbutils`, the system functions `-.POSIXt`, `+.POSIXt`, and `difftime`, get silently replaced by equivalents that force all calculations to be done in seconds. Actually, you **can** still force `difftime` to calculate in different units, via its `units` argument, but the default is now "seconds" rather than "auto".

Usage

```
Hours( x)
Minutes( x)
Seconds( x)
```

Arguments

x numeric vector in the units of the name of the called function.

Value

A numeric vector of the same length as x, giving the equivalent timespan in seconds, together with an attribute `unit="seconds"`.

Author(s)

Mark Bravington

 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`. However, 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. Set the argument `intern=F` if you want to see progress on-screen (but the result won't be captured); if you can handle the disconcertingly blank wait, set `intern=T` to get a character vector result. `set.rcmd.vars` does nothing (yet).

Usage

```
install.pkg( pkg, character.only=FALSE, dir.above.source='+', lib=.libPaths()[1], intern=TRUE)
build.pkg( pkg, character.only=FALSE, dir.above.source='+', intern=TRUE)
build.pkg.binary( pkg, character.only=FALSE, dir.above.source='+', intern=TRUE)
check.pkg( pkg, character.only=FALSE, dir.above.source='+', intern=TRUE)
set.rcmd.vars( ...) # not yet implemented. If you need to set env vars eg PATH for R CMD to work, you have
```

Arguments

pkg	usually an unquoted package name, but interpretation can be changed by non-default <code>character.only</code>
character.only	default FALSE. If TRUE, treat <code>pkg</code> as a normal object, which should therefore be a string containing the package's name. If <code>character.only</code> is itself a string, it will override <code>pkg</code> and be treated as the name of the package.
dir.above.source	where to look for source package; see pre.install
intern	?return the result as a character vector? (See system) Set to FALSE if you want to see the output as-it-happens, but in that case it won't be returned.
lib	where to install to
...	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.)

`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). I should probably allow control of the plethora of checks via appropriate flags, perhaps via the pre-install-hook somehow. However, my understanding that CRAN insists on the full monty, regardless of what you think makes sense for your package. [It *may* also be possible to do some checks directly from R via functions in the **utils** package, but NB the possibility of interference with your current R session; for example, `codoc` attempts to unload & load the package.]

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).

Value

If intern=TRUE: the stuff printed out, with class cat so it prints nicely. If intern=FALSE: various things about the paths (purely for my programming convenience).

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)

## End(Not run)
```

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>>) mlo
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
```

make.NAMESPACE	<i>Auto-create a NAMESPACE file</i>
----------------	-------------------------------------

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

<code>env</code>	character or numeric position on search path
<code>path</code>	directory where proto-package lives
<code>description</code>	(character) elements for the <code>DESCRIPTION</code> file, e.g. <code>c(..., Author="R.A. Fisher", ...)</code> . By default, read from existing file.
<code>more.exports</code>	(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](#)

mcut

Put reals and integers into specified bins, returning factors.

Description

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

Usage

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

Arguments

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

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+
```

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 `mlazied` 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 <code>mtidy</code> and <code>demlazy</code> , defaults to all currently-cached objects in <code>envir</code>
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 than 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_level)` 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=mcaches())` 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)`. `mlocal` functions can be nested, though this gets confusing. By default, all evaluation will happen in the same frame.

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.

Frame-dependent functions (`sys.parent()`) etc. will not do what you expect inside an `mlocal` function. In R 1.8 at least, you need to shift the normal index by 3 to get what you'd expect, so that `sys.call(-3)` inside an `mlocal` function will return the call to the `mlocal` function, and `sys.function(sys.parent(3))` will return the `mlocal` function definition. You can get the expected results for the **caller** via `mvb.sys.parent` with no shift, etc.— unless the caller is itself an `mlocal` function.

`on.exit` **doesn't** work properly, **contrary** to what previous `mlocal` documentation said (the scoping is wrong, though often this doesn't matter). 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.

See R-news 2001 #3 (1/3) for another closely related approach to "macros".

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

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...
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	<i>Replacement and insertion functions with more/less than 1 replacement per spot</i>
----------	---

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.match.call(definition = sys.function(mvb.sys.parent()),
  call = sys.call(mvb.sys.parent()), expand.dots = TRUE)
mvb.nargs()
mvb.sys.call(which = 0)
mvb.sys.function(n)
mvb.sys.on.exit()
```

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.)
<code>which</code>	the number of frame generations to go back.
<code>definition</code>	a function, by default the function from which <code>match.call</code> is called.
<code>call</code>	an unevaluated call to the function specified by <code>definition</code> , as generated by <code>call</code> .
<code>expand.dots</code>	logical. Should arguments matching <code>...</code> in the call be included or left as a <code>...</code> argument?

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`.

Value

See the helpfiles for the system functions.

Author(s)

Mark Bravington

See Also

sys.parent, sys.nframe, parent.frame, match.call, nargs, sys.call, sys.function, sys.on.exit

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
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

```

Arguments

a, b, vector, condition, x, y, what, patt, from, to, cond
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% etc	logical
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% 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'd 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

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 %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)
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
```

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 first thing is that you will need to edit the DESCRIPTION file. `mvbutils` creates a default text file "DESCRIPTION" in the task folder even if you haven't, but it won't be what you really want, as you'll realize if you type `library(help=Splendid)`. Apart from the obvious changes, 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 should rarely need to be updated, since the `autoversion` feature can be used to take care of version numbering.

The additional steps you'll likely need are these:

- Provide documentation (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 should never need to `cd` directly into "Splendid" again.

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`.

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

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 conven-

tional structure of R help if you are really in a hurry. However, the easiest way to add "proper" 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 whether it works. To test the ultimate appearance, you'll need to run `patch.install` to update the help system, as explained below in MAINTAINING.A.PACKAGE. 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 is namespaced (see next), which it must be for R 2.14 up.

Namespaces

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.

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.

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, 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 check specific aspects of an *installed* package, without needing a full-on (and slowish) R CMD CHECK—**but** note that they will first unload your package and then reload it, so they may disrupt your session especially if compiled code is involved. I find `codoc` and `undoc` useful, eg via `codoc("debug", file.path(tasks["debug"], "debug"))`. Nothing is printed unless a problem is found, so a blank result is good news!

Different r versions

You might need to distribute different versions of your package to go with different R versions. (This happened with— at least— 2.10, 2.12, and 2.14.) The `dir . above . source` argument of `pre.install` can be used to create different source package versions. Presumably you'll install the results into different R libraries.

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)
## S3 method for class 'POSIXct'
cbind( ...) # cbind.POSIXct
clip( x, n=1)
cq( ...)
deparse.names.parsably( x)
empty.data.frame( ...)
env.name.string( env)
exists.mvb( x, pos = -1, envir = pos.to.env(pos), frame, mode = "any", inherits = FALSE)
expanded.call( nlocal=sys.parent())
everything( x, by=1, from=1)
find.funs(pos=1, ..., exclude.mcache = TRUE, mode="function")
find.lurking.envs(obj, delve=FALSE, trace=FALSE)
generic.dll.loader( libname, pkgname)
index( lvector)
is.dir( dir)
legal.filename( name)
## S3 replacement method for class 'POSIXct'
length(x) <- value # length<-.POSIXct
lsall( ...)
lib.pos() # used only to hack 'library'
masked( pos)
masking( pos=1)
mkdir( dirlist)
most.recent( lvec)
my.all.equal( x, y)
named( x)
nscat( fmt, ..., sep='\n', file='')
option.or.default( opt.name, default=NULL)
pos( substrs, mainstrs, any.case = FALSE, names.for.output)
## S3 method for class 'cat'
print( x, ...) # print.cat
## S3 method for class 'nullprint'
print( x, ...) # print.nullprint

```

```

put.in.session( ...)
## S3 method for class 'POSIXct'
rbind( ...) # rbind.POSIXct
returnList( ...)
safe.rbind( df1, df2)
scatn( fmt, ..., sep='\n', file='')
to.regexpr( x)
yes.no( prompt, default)

```

Arguments

`x`, `y`, `n`, ..., `value`, `by`, `env`, `from`, `exclude.mcache`, `nlocal`, `lvector`, `dir`, `name`, `pos`, `envir`, `frame`,
see "Arguments by function"

Details

`as.cat` makes a character vector print as if it was catted rather than printed.

`cbind.POSIXct` and `rbind.POSIXct` cope nicely with POSIXct matrices and arrays; default R behaviour is to strip the attributes and turn everything back into raw seconds...

`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— see also `safe.rbind`.

`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.

`exists.mvb` is pretty much like `exists` but has a `pos` rather than a `where` argument.

`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.

`generic.dll.loader` is to be called from the `.onLoad` or `.First.lib` of a package. It looks for its DLLs either directly in the "libs" directory, or in the appropriate sub-architecture directory below "libs".

`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.

`is.dir` tests for directoriness.

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

`length<- .POSIXct` resets the length of POSIXct objects without losing their POSIXity.

`lib.pos` is used by `mvbutils` in `hacking library`, to make sure packages get attached **below** ROOT task. You'll probably never need to call this directly.

`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`: see `scatn`

`option.or.default` returns the named option value if it exists, otherwise the supplied default. Obsolete in R2.10; use `getOption(..., default=...)` 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.

`print`: if `class(x)=="cat"`, the character vector `x` will be printed by `cat(x, sep="\n")`. If `class(x)=="nullprint"`, then `print(x)` will not print anything.

`rbind.POSIXct`: see `cbind.POSIXct`.

`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` 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.

`scatn` is just `cat(sprintf(fmt, ...), "", file=file, sep=sep)`. `scatn` prints a newline afterwards, but not before; `nscat` does the opposite. 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
cbind.POSIXct	matrix of class POSIXct, with same tzone as first element of ...
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"
generic.dll.loader	a useless list
is.dir	logical vector
is.nonzero	TRUE or FALSE
legal.filename	character(1)
length<- .POSIXct	vector of class POSIXct
masked	character vector
masking	character vector
mkdir	logical vector of success/failure
nscat	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
rbind.POSIXct	matrix of class POSIXct, with same tzone as first element of ...
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")`

cbind.POSIXct ...: each element is a matrix/array/vector of class POSIXct.

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

exists.mvb x: string, pos = integer, envir = environment, frame = frame number, mode & inherits as for exists

expanded.call nlocal: frame to retrieve arguments from. Normally, use the default; see [mlocal](#).

everything 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

generic.dll.loader libname, pkgname: as per .onLoad

index lvector: vector of TRUE/FALSE/NA

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

legal.filename name: character string to be modified

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

length<-.POSIXct x: POSIXct object

length<-.POSIXct value: integer

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

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

nscat 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

print.cat, print.nullprint x thing to print

print.cat, print.nullprint ... args for print (ignored)

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

rbind.POSIXct ...: each element is a matrix/array/vector of class POSIXct.

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

```
x <- matrix( Sys.time()+Seconds( 1:6), 2, 3)
rbind( x, x)
cbind( x, x)
length( x) <- 5
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")
everything( 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)
#
#1 what size
#1 attr(obj, "source") 5368
#2 obj 49556
#3 environment(obj) <: namespace:mvbutils> Inf
eapply( .GlobalEnv, find.lurking.envs)
## Not run:
mypack:::onLoad <- function( libname, pkgname) generic.dll.loader( libname, pkgname)
#... or just copy the code into your .onLoad

## End(Not run)
```

```

is.dir( getwd()) # 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)
option.or.default( "my.option", 5) # probably 5
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)
levels( rbind( data.frame( x=1), data.frame( x="cat"))$x)
# NULL, because "x" acquires mode "character"; a bug, I think
levels( safe.rbind( data.frame( x=1), data.frame( x="cat"))$x)
# c( "1", "cat")
scatn( 'Things %i', 1:3)
to.regexpr( "a{f}") # "a\\{\\{f"
## 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 `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 concatenation 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

```
#e <- new.env()
#evalq({
#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
```

my.package.path	<i>Given a function in a loaded package, find the path of the package.</i>
-----------------	--

Description

Useful for finding the path of special files in a package. Usually this will be called inside a function which lives in a package. If the package has a namespace, the path object stored in the `.__NAMESPACE__` environment is used; if not, the path attribute of the attached package in the search path is used. The package has to be loaded, of course. Doesn't work for things in `baseenv()`.

Usage

```
my.package.path( func=sys.function( sys.parent()))
```

Arguments

func	a function in a package. If you're calling it from a function in that package, the default will usually do.
------	---

Value

Path: a character string, or NULL if not found.

Strange usage

If you are doing something weird, then the default func might not you work. Instead, you can set func manually beforehand, e.g.:

```
me <- sys.function()
my.path <- lapply( list( me), my.package.path)[[1]]
```

Examples

```
## Not run:
my.package.path( ks.test)
mypack:::show.description.file <- function() as.cat( readLines(
  file.path( my.package.path(), 'DESCRIPTION'))))
## End(Not run)
```

pre.install

*Update a source and/or installed package from a task package***Description**

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 OVERRIDING.DEFAULTS below.

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

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=option.or.default("mvb.update.installed.cache", TRUE),
  pre.inst=TRUE, dir.above.source="+", R.target.version=getRversion(),
  autoversion=getOption("mvb.autoversion", TRUE))
patch.install(...) # actually the args are exactly the same as for 'patch.installed'
```

Arguments

`pkg` package name. Either quoted or unquoted is OK; unquoted will be treated as quoted unless...

`character.only` ... is 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; if TRUE, then recreate help for all documented objects.
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? Leave as TRUE 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 file. 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 subdirectory "`<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" subdirectory, possibly a "sysdata.rda" file in the same place to contain non-functions, and a set

of Rd files in the "man" subdirectory. 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" subdirectory of your task, which get copied directly into the package. Any "inst", "demo", "tests", "src", "exec", and "data" subdirectories will be copied to the source package, recursively (i.e. including any of *their* subdirectories). 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`, `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" subdirectories of the source-package directory will be removed (unless you have some [mlazy](#) objects; see below). If an `.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" subdirectory of the package, as will any files in the task's own "inst" subdirectory. Similarly, any DESCRIPTION file in the task package will be copied to the source package, after removing any "Built:" line. If there is no DESCRIPTION file in the task package, a default DESCRIPTION file will be created in the package directory, but you'll certainly want to edit it before CRAN release; you can also generate the DESCRIPTION file yourself via the `pre.install.hook` override. Any "Makefile.*" in the task package will be copied, as will any in the "src" subdirectory (not sure why both places are allowed). No other files or subdirectories 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" subdirectory, which is subsequently used by `patch.installed`. `build.pkg` (or R CMD BUILD) 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" subdirectory of the installed package, without needing to re-install. I haven't tried this, but colleagues have.

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`.

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" subdirectory. Data lives either in the "sysdata.rdb/rdx" files in the "R" subdirectory (but can still be user-visible, which is not normally the case for objects in those files), or in the "mlazy" subdirectory 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)`. 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!

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: 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.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 subdirectory 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`

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 `subdir` 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` \geq 2.14, regardless of whether your package has a `.onLoad` or not.

If you are using both `R < 2.12` and `R \geq 2.12`, then bear in mind the `subdir` argument of `pre.install`, which lets you create different source packages; this is useful with precompiled DLLs, since the source package structure needs to be different.

Help pre r2 10: For help conversion to work, the various R build tools must be in the search path, just as when you're actually building the package. Also, certain environment variables may need to be set, such as `"R_LIBS"`. All this may not automatically be the case. If not, you should set `options(rcmd.shell.setup)` to a character vector of commands that will set up the path & environment variables properly, when called as part of a batch file (Windows) or shell script (Linux etc). On my (Windows) system, I wrote a batch file which I invoke via `CALL SET-R-BUILD-PATH-GUTS.BAT`. The changes are temporary, just while the conversion is taking place.

Compiled code on windows pre r2 12: There didn't use to be an "«arch»" level, so that DLLs went directly into `"inst/libs"` (task and source package) and `"libs"` (installed package).

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 \leq 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
lbrary( mypack) # maybe optional
# ..do various things involving changes to mypack
patch.install( mypack) # keep disk image up-to-date
# Prepare copies for distribution
maintain.packages( mypack) # only once per session
# library( mypack) is optional
build.pkg( mypack) # for Linux or CRAN
build.pkg.binary( mypack) # for Windows or Macs

## End(Not run)
```

readLines.mvb

Read text lines from a connection

Description

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

Usage

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

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.

Details

Apart from stopping if the EOF line is encountered, behaviour should be as for readLines.

Value

A character vector of length 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. glurb

Usage

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

Arguments

<code>pkg</code>	(string, or environment) package name or environment, e.g. <code>..mypack</code>
<code>...</code>	unquoted object names to remove
<code>list</code>	character vector alternative to <code>...</code> , which is ignored if <code>list</code> is set

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 `i`th 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

<code>pos</code>	string or numeric position on search path, or environment (e.g. <code>..mypack</code> if "mypack" is a maintained-package).
<code>path</code>	directory or file to save into (see Details).
<code>ascii</code>	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 whose source contains a regexp. Can also search flatdoc-style function doco and character doc objects.*

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".

Usage

```
search.for.regexpr( pattern, where=1, lines=FALSE, doc=FALSE, ...)
```

Arguments

pattern	the regexp
where	an environment, something that can be coerced to an environment (so the default corresponds to <code>.GlobalEnv</code>), 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. If TRUE, search the usual flatdoc places, i.e. "doc" attributes of functions, and character objects whose name ends in ".doc".
...	passed to <code>grep</code> — 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](#)

setup.mcache	<i>Cacheing objects for lazy-load access</i>
--------------	--

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("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("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](#)

source.mvb	<i>Read R code and data from a file or connection</i>
------------	---

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, prompt.echo, evaluate=TRUE, debug.script=
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.
echo	logical; defaults to option.or.default("verbose", FALSE)
prompt.echo	what to show before the first line of each echoed command.
evaluate	TRUE if you want the text to be evaluated, rather than just parsed expression-by-expression (in which case, the last parsed statement is returned).
debug.script	FALSE, unless TRUE if you want to debug the script itself. Requires the debug package.
EOF	line which terminates data block; lines afterwards will again be treated as R statements.
...	other args to read.table

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 use `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.

To do

Add at least the `local=FALSE` argument of `source`.

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, the DESCRIPTION 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. 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.

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	<i>Extract subset and warn about omitted cases</i>
-----------------	--

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

<code>x</code>	<code>data.frame</code>
<code>cond</code>	expression to evaluate in the context of <code>data.frame</code> . If <code>sub=TRUE</code> (the default), this will be substituted, so <code>.</code> . If <code>sub=FALSE</code> , you can use a pre-assigned expression; in that case, you had better set <code>mess.cond</code> manually.
<code>mess.head</code>	description of <code>data.frame</code> (e.g. its name) for use in a warning.
<code>mess.cond</code>	description of the desired condition for use in a warning.
<code>row.info</code>	character vector that will describe rows; omitted elements appear in the warning
<code>sub</code>	should <code>cond</code> 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

- *Topic **IO**
 - readLines.mvb, 80
- *Topic **data**
 - mlazy, 47
 - setup.mcache, 85
- *Topic **debugging**
 - Save, 82
- *Topic **documentation**
 - dochelp, 20
 - find.documented, 24
 - flatdoc, 31
 - get.backup, 35
- *Topic **file**
 - Save, 82
- *Topic **misc**
 - changed.funs, 13
 - doc2Rd, 15
 - dont.lockBindings, 22
 - fast.read.fwf, 24
 - foodweb, 33
 - install.pkg, 40
 - lsize, 44
 - mcut, 46
 - multirep, 55
 - mvbutils-package, 2
 - mvbutils.operators, 57
 - mvbutils.packaging.tools, 59
 - mvbutils.utils, 64
 - my.package.path, 72
 - rm.pkg, 81
 - search.for.regexpr, 84
 - source.mvb, 85
 - unpackage, 90
 - warn.and.subset, 91
- *Topic **programming**
 - do.in.envir, 13
 - dont.lock.me, 21
 - extract.named, 23
 - find.documented, 24
 - fixr, 27
 - flatdoc, 31
 - get.backup, 35
 - hack, 38
 - help2flatdoc, 39
 - local.on.exit, 42
 - local.return, 43
 - lsize, 44
 - make.NAMESPACE, 45
 - mlazy, 47
 - mlocal, 51
 - mvb.sys.parent, 56
 - my.index, 70
 - pre.install, 73
 - setup.mcache, 85
 - strip.missing, 88
- *Topic **utilities**
 - cd, 6
 - cdfind, 10
 - cdprompt, 12
 - do.in.envir, 13
 - find.documented, 24
 - fix.order, 26
 - fixr, 27
 - get.backup, 35
 - Hours, 40
 - make.NAMESPACE, 45
 - move, 53
 - my.index, 70
 - pre.install, 73
 - task.home, 89
 - %!in% (mvbutils.operators), 57
 - %**% (mvbutils.operators), 57
 - %SUCH.THAT% (mvbutils.operators), 57
 - %&% (mvbutils.operators), 57
 - %downto% (mvbutils.operators), 57
 - %except% (mvbutils.operators), 57
 - %grepling% (mvbutils.operators), 57
 - %in.range% (mvbutils.operators), 57

- `%is.a%` (mvbutils.operators), 57
- `%is.an%` (mvbutils.operators), 57
- `%is.not.a%` (mvbutils.operators), 57
- `%is.not.an%` (mvbutils.operators), 57
- `%matching%` (mvbutils.operators), 57
- `%not.in%` (mvbutils.operators), 57
- `%perling%` (mvbutils.operators), 57
- `%such.that%` (mvbutils.operators), 57
- `%that.are.in%` (mvbutils.operators), 57
- `%that.dont.match%` (mvbutils.operators), 57
- `%that.match%` (mvbutils.operators), 57
- `%upto%` (mvbutils.operators), 57
- `%where.warn%` (mvbutils.operators), 57
- `%where%` (mvbutils.operators), 57
- `%without.name%` (mvbutils.operators), 57

- `as.cat` (mvbutils.utils), 64
- `assign.to.base` (hack), 38
- `attach.mlazy` (mlazy), 47
- `autoedit` (fixr), 27

- `build.pkg`, 61–63, 75
- `build.pkg` (install.pkg), 40
- `build.pkg.binary`, 61, 63

- `callees.of` (foodweb), 33
- `callers.of` (foodweb), 33
- `cbind.POSIXct`, 4, 5, 66
- `cbind.POSIXct` (mvbutils.utils), 64
- `cd`, 3–5, 6, 6, 10–12, 28, 30, 37, 47–50, 53, 54, 60, 61, 74, 77, 79, 82, 85, 89, 90
- `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.pkg`, 63
- `check.pkg` (install.pkg), 40
- `clip` (mvbutils.utils), 64
- `cq` (mvbutils.utils), 64
- `create.backups` (get.backup), 35

- `current.source`, 81
- `current.source` (source.mvb), 85
- `cut`, 46

- `data`, 77
- `demlazy` (mlazy), 47
- `deparse.names.parsably` (mvbutils.utils), 64
- `do.in.envir`, 6, 13, 43, 52
- `doc2Rd`, 15, 21, 25, 32, 39, 60, 62, 77, 79
- `dochelp`, 4, 6, 16, 17, 20, 25, 32
- `dont.lock.me`, 21
- `dont.lockBindings`, 22

- `empty.data.frame` (mvbutils.utils), 64
- `env.name.string` (mvbutils.utils), 64
- `everyth` (mvbutils.utils), 64
- `exists.mvb` (mvbutils.utils), 64
- `expanded.call` (mvbutils.utils), 64
- `extract.named`, 3, 23

- `fast.read.fwf`, 24
- `FF`, 32, 36, 82
- `FF` (fixr), 27
- `find.docholder`, 84
- `find.docholder` (find.documented), 24
- `find.documented`, 21, 24, 84
- `find.funs` (mvbutils.utils), 64
- `find.lurking.envs`, 63
- `find.lurking.envs` (mvbutils.utils), 64
- `fix.order`, 26, 30, 36
- `fixr`, 3, 5–8, 10, 15, 26, 27, 31, 32, 35–37, 60–62
- `fixtext` (fixr), 27
- `flatdoc`, 6, 19, 21, 25, 29, 31, 39, 46, 61, 62, 75, 77, 81, 84, 87
- `foodweb`, 3, 6, 11, 33
- `from.here` (source.mvb), 85

- `generic.dll.loader`, 62
- `generic.dll.loader` (mvbutils.utils), 64
- `get.backup`, 30, 35

- `hack`, 38
- `hack` (hack), 38
- `help2flatdoc`, 16, 19, 39
- `Hours`, 3, 40

- `index` (mvbutils.utils), 64
- `install.pkg`, 40, 61, 62, 73, 76

- is.dir (mvbutils.utils), 64
- legal.filename (mvbutils.utils), 64
- length<- .POSIXct (mvbutils.utils), 64
- lib.pos (mvbutils.utils), 64
- load.refdb, 50
- local.on.exit, 42, 52
- local.return, 43, 43, 52
- localfuncs, 52
- lsall (mvbutils.utils), 64
- lsize, 44, 50
- maintain.packages, 6, 21, 28, 54, 60–62, 73, 79, 81
- make.arguments.section, 32
- make.NAMESPACE, 45, 75
- make.usage.section, 32, 77
- makeActiveBinding, 50
- masked (mvbutils.utils), 64
- masking (mvbutils.utils), 64
- massrep (multirep), 55
- mcachees (mlazy), 47
- mcut, 3, 46
- mintcut (mcut), 46
- Minutes (Hours), 40
- mkdir (mvbutils.utils), 64
- mlazy, 3, 5, 6, 9, 10, 44, 47, 68, 75, 77, 82, 83, 85
- mlocal, 3, 6, 14, 38, 42–44, 51, 68
- most.recent (mvbutils.utils), 64
- move, 5, 10, 28, 30, 36, 37, 49, 53, 62, 82, 85
- move (move), 53
- mtidy (mlazy), 47
- multinsert (multirep), 55
- multirep, 3, 55
- mvb.match.call (mvb.sys.parent), 56
- mvb.nargs (mvb.sys.parent), 56
- mvb.parent.frame (mvb.sys.parent), 56
- mvb.sys.call (mvb.sys.parent), 56
- mvb.sys.function (mvb.sys.parent), 56
- mvb.sys.nframe (mvb.sys.parent), 56
- mvb.sys.on.exit (mvb.sys.parent), 56
- mvb.sys.parent, 52, 56
- mvbutils (mvbutils-package), 2
- mvbutils-package, 2
- mvbutils.operators, 3, 6, 57
- mvbutils.packaging.tools, 3, 4, 8, 10, 29, 31, 59, 73, 74, 79, 90
- mvbutils.utils, 3, 6, 64
- my.all.equal (mvbutils.utils), 64
- my.index, 70
- my.index (my.index), 70
- my.index.assign (my.index), 70
- my.index.exists (my.index), 70
- my.package.path, 72
- named (mvbutils.utils), 64
- nscat (mvbutils.utils), 64
- option.or.default (mvbutils.utils), 64
- par, 33
- patch.install, 15, 41, 61, 62
- patch.install (pre.install), 73
- patch.installed, 19
- patch.installed (pre.install), 73
- plot.cdtree (cdfind), 10
- plot.foodweb (foodweb), 33
- pos, 4, 10, 24, 83
- pos (mvbutils.utils), 64
- pre.install, 15, 17–19, 39, 41, 45, 46, 60–64, 73, 90
- print.cat, 19, 29
- print.cat (mvbutils.utils), 64
- print.nullprint (mvbutils.utils), 64
- put.in.session (mvbutils.utils), 64
- rbind.POSIXct, 4, 5
- rbind.POSIXct (mvbutils.utils), 64
- read.bkind (get.backup), 35
- read.table, 24
- readLines.mvb, 80, 86, 87
- readr (fixr), 27
- remove.from.package (rm.pkg), 81
- returnList, 43, 88
- returnList (mvbutils.utils), 64
- rm.pkg, 62, 81
- safe.rbind (mvbutils.utils), 64
- Save, 5, 9, 28, 36, 47–50, 54, 82, 85
- save, 82, 83
- Save.pos, 36, 54, 61
- scatn, 66, 68
- scatn (mvbutils.utils), 64
- search.for.regexpr, 3, 84
- Seconds (Hours), 40
- set.rcmd.vars (install.pkg), 40
- setup.mcache, 85

source.mvb, 3, 6, 32, 81, 85
strip.missing, 3, 88
strwrap, 20
system, 41

task.home, 9, 10, 48, 89
to.regexpr (mvbutils.utils), 64

unpackage, 90
unpackage (unpackage), 90

warn.and.subset, 91
warn.and.subset (warn.and.subset), 91
write.NAMESPACE (make.NAMESPACE), 45
write.sourceable.function, 19, 32

yes.no (mvbutils.utils), 64