Package 'reproducible'

August 26, 2026

Type Package

Title Enhance Reproducibility of R Code

Description A collection of high-level, machine- and OS-independent tools for making reproducible and reusable content in R.
The two workhorse functions are 'Cache()' and 'prepInputs()'.
'Cache()' allows for nested caching, is robust to environments and objects with environments (like functions), and deals with some classes of file-backed R objects e.g., from 'terra' and 'raster' packages.
Both functions have been developed to be foundational components of data retrieval and processing in continuous workflow situations. In both functions, efforts are made to make the first and subsequent calls of functions have the same result, but faster at subsequent times by way of checksums and digesting. Several features are still under development, including cloud storage of cached objects allowing for sharing between users. Several advanced options are available, see '?reproducibleOptions()'.

SystemRequirements 'unrar' (Linux/macOS) or '7-Zip' (Windows) to work with '.rar' files.

URL https://reproducible.predictiveecology.org, https://github.com/PredictiveEcology/reproducible

Date 2026-08-25

Version 3.2.1

Depends R (>= 4.3)

Imports cli, data.table (>= 1.10.4), digest, filelock (>= 1.0.3), fpCompare, fs, lifecycle, lobstr, methods, stats, tools, utils

Suggests archive, covr, curl (>= 7.0.0), DBI, future, geodata, glue, googledrive, httr, httr2 (>= 1.2.1), knitr, parallel, parallelly, qs2, raster (>= 3.5-15), RCurl (>= 1.95-4.8), rlang, rmarkdown, RSQLite, R.utils, rvest, sf, sp (>= 1.4-2), terra (>= 1.7-20), testthat, withr

Encoding UTF-8

Language en-CA

License GPL-3

VignetteBuilder knitr, rmarkdown

BugReports https://github.com/PredictiveEcology/reproducible/issues

ByteCompile yes

Config/roxygen2/version 8.1.0

Collate 'DBI.R' 'cache-call.R' 'cache-digest.R' 'cache-helpers.R'
'cache-internals.R' 'cache-repo.R' 'cache-report.R'
'robustDigest.R' 'messages.R' 'cache.R' 'cacheGeo.R'
'checksums.R' 'cloud.R' 'convertPaths.R' 'copy.R' 'download.R'
'downloadTileAndUpload.R' 'exportedMethods.R' 'gis.R'
'helpers.R' 'listNamed.R' 'objectSize.R' 'options.R'
'packages.R' 'paths.R' 'pipe.R' 'postProcess.R'
'postProcessTo.R' 'preProcess.R' 'prepInputs.R'
'prepInputsCOG.R' 'reproducible-deprecated.R'
'reproducible-package.R' 'search.R' 'showCacheEtc.R'
'spatialObjects-class.R' 'terra-migration.R' 'urlLog.R' 'zzz.R'

NeedsCompilation no

Author Eliot J B McIntire [aut, cre] (ORCID:
<https://orcid.org/0000-0002-6914-8316>),
Alex M Chubaty [aut] (ORCID: <https://orcid.org/0000-0001-7146-8135>),
Tati Micheletti [ctb] (ORCID: <https://orcid.org/0000-0003-4838-8342>),
Ceres Barros [ctb] (ORCID: <https://orcid.org/0000-0003-4036-977X>),
Ian Eddy [ctb] (ORCID: <https://orcid.org/0000-0001-7397-2116>),
His Majesty the King in Right of Canada, as represented by the Minister
of Natural Resources Canada [cph]

Maintainer Eliot J B McIntire <eliot.mcintire@canada.ca>

Repository CRAN

Date/Publication 2026-08-25 22:40:02 UTC

Contents

reproducible-package 4
.addTagsRepo 5
.debugCache 7
.file.move 8
.isGridded 8
.isMemoised 9
.objSizeWithTry 10
.prefix 10
.prepareFileBackedRaster 11
.removeCacheAtts 12
.requireNamespace 13
.setSubAttrInList 13
.whereInStack 14
.wrap 14
assessDataType 18

basename2 . . . 21
Cache . . . 21
CacheDigest . . . 32
CacheGeo . . . 33
cacheId . . . 36
checkPath . . . 37
checkRelative . . . 38
Checksums . . . 39
cloudDownload . . . 41
compareNA . . . 42
convertCallToCommonFormat . . . 43
convertPaths . . . 43
Copy . . . 44
copySingleFile . . . 46
createCache . . . 48
detectActiveCores . . . 52
determineFilename . . . 52
downloadRemote . . . 54
extractFromArchive . . . 56
Filenames . . . 57
fixErrorsIn . . . 58
gdalProject . . . 59
getRelative . . . 61
harmonizeCall . . . 62
isUpdated . . . 63
keepOrigGeom . . . 64
linkOrCopy . . . 64
listGoogleDriveFolder . . . 66
listNamed . . . 67
makeUrlRemap . . . 68
matchCall2 . . . 69
mergeCache . . . 70
messageDF . . . 71
minFn . . . 73
movedCache . . . 74
normPath . . . 75
numCoresToUse . . . 77
objSize . . . 78
paddedFloatToChar . . . 79
Path-class . . . 80
postProcess . . . 82
postProcessInputsSuperseded . . . 85
postProcessTo . . . 87
prepInputs . . . 92
prepInputsLog . . . 98
prepInputsWithTiles . . . 99
prepopulateCacheAsync . . . 101
preProcessParams . . . 102

purgeChecksums . . . 105
rasterRead . . . 106
remapFilenames . . . 106
reproducibleOptions . . . 107
retry . . . 115
saveToCache . . . 116
searchFull . . . 117
set.randomseed . . . 118
showCache . . . 119
studyAreaName . . . 123
tempdir2 . . . 124
tempfile2 . . . 125
unrarPath . . . 126
usesPointer . . . 126
writeFuture . . . 127

Index **129**

`reproducible-package` *The* `reproducible` *package*

Description

[Stable]

This package aims at making high-level, robust, machine and OS independent tools for making deeply reproducible and reusable content in R. The core user functions are `Cache` and `prepInputs`. Each of these is built around many core and edge cases required to have reproducible code of arbitrary complexity.

Main Tools

There are many elements within the reproducible package. However, there are currently two main ones that are critical for reproducible research. The key element for reproducible research is that the code must always return the same content every time it is run, but it must be vastly faster the 2nd, 3rd, 4th etc, time it is run. That way, the entire code sequence for a project of arbitrary size can be run *from the start* every time.

`Cache()`**:** A robust wrapper for any function, including those with environments, disk-backed storage (currently on `Raster`) class), operating-system independent, whose first time called will execute the function, second time will compare the inputs to a database of entries, and recover the first result if inputs are identical. If `options("reproducible.useMemoise" = TRUE)`, the second time will be very fast as it will recover the answer from RAM.

`prepInputs()`**for other specifics for other classes.:** Download, or load objects, and possibly post-process them. The main advantage to using this over more direct routes is that it will automatically build checksums tables, use `Cache` internally where helpful, and possibly run a variety of post-processing actions. This means this function can also itself be cached for even more speed. This allows all project data to be stored in custom cloud locations or in their original online data repositories, `without altering code` between the first, second, third, etc., times the code is run.

Package options

See reproducibleOptions() for a complete description of package options() to configure behaviour.

Author(s)

Maintainer: Eliot J B McIntire <eliot.mcintire@canada.ca> (ORCID)

Authors:

- Eliot J B McIntire <eliot.mcintire@canada.ca> (ORCID)
- Alex M Chubaty <achubaty@for-cast.ca> (ORCID)

Other contributors:

- Tati Micheletti <tati.micheletti@gmail.com> (ORCID) [contributor]
- Ceres Barros <ceres.barros@ubc.ca> (ORCID) [contributor]
- Ian Eddy <ian.eddy@nrcan-rncan.gc.ca> (ORCID) [contributor]
- His Majesty the King in Right of Canada, as represented by the Minister of Natural Resources Canada [copyright holder]

See Also

Useful links:

- https://reproducible.predictiveecology.org
- https://github.com/PredictiveEcology/reproducible
- Report bugs at https://github.com/PredictiveEcology/reproducible/issues

---

.addTagsRepo *Add a Tag to a Cached Object in the Repository*

Description

This hidden function appends a single tag (key-value pair) to the metadata of a cached object identified by its cacheId. Tags can be stored either in a database (via DBI) or in a file-based cache system.

Updates the value of an existing tag for a cached object identified by its cacheId. If the tag does not exist and add = TRUE, the tag will be added. This function supports both database-backed and file-based cache systems.

Usage

```
.addTagsRepo(
  cacheId,
  cachePath = getOption("reproducible.cachePath"),
  tagKey = character(),
  tagValue = character(),
  cacheSaveFormat = getOption("reproducible.cacheSaveFormat"),
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose")
)

.updateTagsRepo(
  cacheId,
  cachePath = getOption("reproducible.cachePath"),
  tagKey = character(),
  tagValue = character(),
  add = TRUE,
  cacheSaveFormat = getOption("reproducible.cacheSaveFormat"),
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose")
)
```

Arguments

cacheId	character(1) Unique identifier of the cached object. Must be of length 1.
cachePath	character(1) Path to the cache directory. Defaults to getOption("reproducible.cachePath").
tagKey	character(1) The key for the tag. Must be supplied.
tagValue	character(1) The new value for the tag. Must be supplied.
cacheSaveFormat	character(1) Format used for saving cache files. Defaults to getOption("reproducible.cacheSaveFo
drv	A DBI driver object. Defaults to getDrv(getOption("reproducible.drv", NULL)).
conn	A DBI connection object. If NULL, a new connection is created internally.
verbose	integer or logical. If > 0 or TRUE, print informational messages. Defaults to getOption("reproducible.verbose").
add	logical(1) If TRUE, adds the tag if it does not exist. Defaults to TRUE.

Details

This function is primarily used internally by the reproducible package to maintain metadata about cached objects. It supports both database-backed and file-based caching systems.

- If useDBI() returns TRUE, the tag update is performed in the database table.
- If no rows are affected and add = TRUE, the tag is inserted using .addTagsRepo().
- For file-based caches, the function modifies the tag in the corresponding metadata file.

Value

NULL (invisibly). The function is called for its side effects.

NULL (invisibly). Called for its side effects.

See Also

.addTagsRepo() for adding tags without updating.

Examples

```
a <- Cache(rnorm(1))
.addTagsRepo(cacheId = gsub("cacheId:", "", attr(a, "tags")),
             tagKey = "status", tagValue = "processed")
showCache() # last entry is the above line

a <- Cache(rnorm(1))
# Update an existing tag
.updateTagsRepo(cacheId = gsub("cacheId:", "", attr(a, "tags")),
             tagKey = "status", tagValue = "second")

# Add a tag if it doesn't exist
.updateTagsRepo(cacheId = gsub("cacheId:", "", attr(a, "tags")),
             tagKey = "status", tagValue = "new", add = TRUE)
```

.debugCache *Attach debug info to return for Cache*

Description

Internal use only. Attaches an attribute to the output, usable for debugging the Cache.

Usage

```
.debugCache(obj, preDigest, ..., fullCall)
```

Arguments

obj	An arbitrary R object.
preDigest	A list of hashes.
...	Dots passed from Cache
fullCall	The original call to Cache

Value

The same object as obj, but with 2 attributes set.

Author(s)

Eliot McIntire

`.file.move` *Move a file to a new location – Defunct – use* `hardLinkOrCopy`

Description

This will first try to `file.rename`, and if that fails, then it will `file.copy` then `file.remove`.

Usage

```
.file.move(from, to, overwrite = FALSE)
```

Arguments

`from`, `to`	character vectors, containing file names or paths.
`overwrite`	logical indicating whether to overwrite destination file if it exists.

Value

Logical indicating whether operation succeeded.

`.isGridded` *Some spatial helper functions*

Description

Some spatial helper functions

Usage

```
.isGridded(x)

.isVector(x)

.isSF(x)

.isSpat(x)

.isSpatialAny(x)

.isCRSany(x)
```

Arguments

x	A spatial object.

Details

`.isGridded` returns `TRUE` if the object is a `SpatRaster` or `Raster`

`.isVector` returns `TRUE` if the object is `SpatVector`, `spatial` or `sf`

`.isSF` returns `TRUE` if the object is `sf` or `sfc`

`.isSpat` returns `TRUE` if the object is `SpatVector` or `SpatRaster`

`.isSpatialAny` returns `TRUE` if the object returns `TRUE` for `.isGridded` or `.isVector`

Value

Logical.

`.isMemoised` *Evaluate whether a cacheId is memoised*

Description

Intended for internal use. Exported so other packages can use this function.

Usage

```
.isMemoised(cacheId, cachePath = getOption("reproducible.cachePath"))
```

Arguments

`cacheId`	Character string. If passed, this will override the calculated hash of the inputs, and return the result from this `cacheId` in the `cachePath`. Setting this is equivalent to manually saving the output of this function, i.e., the object will be on disk, and will be recovered in subsequent This may help in some particularly finicky situations where `Cache` is not correctly detecting unchanged inputs. This will guarantee the object will be identical each time; this may be useful in operational code.
`cachePath`	A repository used for storing cached objects. This is optional if `Cache` is used inside a SpaDES module.

Value

A logical, length 1 indicating whether the `cacheId` is memoised.

.objSizeWithTry lobstr::obj_size *with a* try *to address issue #72*

Description

It is not clear why, but it appears that running `lobstr::obj_size` again, after a bad binding error, it will work.

Usage

```
.objSizeWithTry(x, useTry = TRUE)
```

Arguments

x	An object
useTry	Logical. If TRUE, the default, then it will use try. Can optionally avoid this if set to FALSE. The try takes sufficient extra compute time that it is worth avoiding it if possible.

Value

The size of an object, using `lobstr::obj_size` or `object.size` if the first fails

.prefix *Add a prefix or suffix to the basename part of a file path*

Description

Prepend (or postpend) a filename with a prefix (or suffix). If the directory name of the file cannot be ascertained from its path, it is assumed to be in the current working directory.

Usage

```
.prefix(f, prefix = "")

.suffix(f, suffix = "")
```

Arguments

f	A character string giving the name/path of a file.
prefix	A character string to prepend to the filename.
suffix	A character string to postpend to the filename.

Value

A character string or vector with the prefix pre-pended or suffix post-pended on the basename of the f, before the file extension.

Author(s)

Jean Marchal and Alex Chubaty

Examples

```
# file's full path is specified (i.e., dirname is known)
myFile <- file.path("~/data", "file.tif")
.prefix(myFile, "small_") ## "/home/username/data/small_file.tif"
.suffix(myFile, "_cropped") ## "/home/username/data/myFile_cropped.shp"

# file's full path is not specified
.prefix("myFile.shp", "small") ## "./small_myFile.shp"
.suffix("myFile.shp", "_cropped") ## "./myFile_cropped.shp"
```

.prepareFileBackedRaster

Copy the file-backing of a file-backed Raster object*

Description

Rasters are sometimes file-based, so the normal save and copy and assign mechanisms in R don't work for saving, copying and assigning. This function creates an explicit file copy of the file that is backing the raster, and changes the pointer (i.e., filename(object)) so that it is pointing to the new file.

Usage

```
.prepareFileBackedRaster(
  obj,
  repoDir = NULL,
  overwrite = FALSE,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  ...
)
```

Arguments

obj	The raster object to save to the repository.
repoDir	Character denoting an existing directory in which an artifact will be saved.
overwrite	Logical. Should the raster be saved to disk, overwriting existing file.
drv	If using a database backend, drv must be an object that inherits from DBIDriver (e.g., RSQLite::SQLite).
conn	an optional DBIConnection object, as returned by dbConnect().
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
...	Not used

Value

A raster object and its newly located file backing. Note that if this is a legitimate Cache repository, the new location will be a subdirectory called 'rasters/' of 'repoDir/'. If this is not a repository, the new location will be within repoDir.

Author(s)

Eliot McIntire

.removeCacheAtts *Remove attributes that are highly varying*

Description

Remove attributes that are highly varying

Usage

```
.removeCacheAtts(x)
```

Arguments

x	Any arbitrary R object that could have attributes

.requireNamespace *Provide standard messaging for missing package dependencies*

Description

This provides a standard message format for missing packages, e.g., detected via `requireNamespace`.

Usage

```
.requireNamespace(
  pkg = "methods",
  minVersion = NULL,
  stopOnFALSE = FALSE,
  messageStart = NULL
)
```

Arguments

`pkg`	Character string indicating name of package required
`minVersion`	Character string indicating minimum version of package that is needed
`stopOnFALSE`	Logical. If `TRUE`, this function will create an error (i.e., `stop`) if the function returns `FALSE`; otherwise it simply returns `FALSE`
`messageStart`	A character string with a prefix of message to provide

Value

A logical or stop if the namespace is not available to be loaded.

.setSubAttrInList *Set subattributes within a list by reference*

Description

Sets only a single element within a list attribute.

Usage

```
.setSubAttrInList(object, attr, subAttr, value)
```

Arguments

`object`	An arbitrary object
`attr`	The attribute name (that is a list object) to change
`subAttr`	The list element name to change
`value`	The new value

Value

This sets or updates the `subAttr` element of a list that is located at `attr(object, attr)`, with the `value`. This, therefore, updates a sub-element of a list attribute and returns that same object with the updated attribute.

`.whereInStack` *Search for objects in the call stack*

Description

Normally, this is only used in special, advanced uses. The standard approach to getting an object from an environment in the call stack is to explicitly pass it into the function.

Usage

```
.whereInStack(obj, startingEnv = parent.frame())
```

Arguments

`obj`	Character string. The object name to search.
`startingEnv`	An environment to start searching in.

Value

The environment in which the object exists. It will return the first environment it finds, searching outwards from where the function is used.

`.wrap` *Deal with class for saving to and loading from Cache or Disk*

Description

This generic and some methods will do whatever is required to prepare an object for saving to disk (or RAM) via e.g., `saveRDS`. Some objects (e.g., `terra`'s `Spat*`) cannot be saved without first wrapping them. Also, file-backed objects are similar.

Usage

```
.wrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  preDigest,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  outputObjects = NULL,
  cacheId = NULL,
  ...
)

## S3 method for class 'list'
.wrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  preDigest,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  outputObjects = NULL,
  cacheId = NULL,
  ...
)

## S3 method for class 'environment'
.wrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  preDigest,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  outputObjects = NULL,
  cacheId = NULL,
  ...
)

## Default S3 method:
.wrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  preDigest,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  outputObjects = NULL,
```

```
  cacheId = NULL,
  ...
)

## Default S3 method:
.unwrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  cacheId = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  ...
)

.unwrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  cacheId = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  ...
)

## S3 method for class 'environment'
.unwrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  cacheId = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  ...
)

## S3 method for class 'list'
.unwrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  cacheId = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  ...
)

## S3 method for class 'PackedSpatExtent2'
.unwrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  cacheId = NULL,
```

```
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  ...
)

## S3 method for class 'PackedSpatVector2'
.unwrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  cacheId = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  ...
)

## S3 method for class 'data.table'
.unwrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  cacheId = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  ...
)

## S3 method for class 'PackedSpatVector'
.unwrap(
  obj,
  cachePath = getOption("reproducible.cachePath"),
  cacheId = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  ...
)
```

Arguments

obj	Any arbitrary R object.
cachePath	A repository used for storing cached objects. This is optional if Cache is used inside a SpaDES module.
preDigest	The list of preDigest that comes from CacheDigest of an object
drv	If using a database backend, drv must be an object that inherits from DBIDriver (e.g., RSQLite::SQLite).
conn	an optional DBIConnection object, as returned by dbConnect().
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

outputObjects	Optional character vector indicating which objects to return. This is only relevant for list, environment (or similar) objects
cacheId	Used strictly for messaging. This should be the cacheId of the object being recovered. Default is NULL.
...	Arguments passed to methods; default does not use anything in ....

Value

Returns an object that can be saved to disk e.g., via saveRDS.

Examples

```
# For SpatExtent
if (requireNamespace("terra")) {
  ex <- terra::ext(c(0, 2, 0, 3))
  exWrapped <- .wrap(ex)
  ex1 <- .unwrap(exWrapped)
}
```

---

assessDataType *Assess the appropriate raster layer data type*

Description

When writing raster-type objects to disk, a datatype can be specified. These functions help identify what smallest datatype can be used.

Usage

```
assessDataType(ras, type = "writeRaster")

## Default S3 method:
assessDataType(ras, type = "writeRaster")
```

Arguments

ras	The RasterLayer or RasterStack for which data type will be assessed.
type	Character. "writeRaster" (default) or "GDAL" (defunct) to return the recommended data type for writing from the raster packages, respectively, or "projectRaster" to return recommended resampling type.

Value

A character string indicating the data type of the spatial layer (e.g., "INT2U"). See terra::datatype()

Examples

```
if (requireNamespace("terra", quietly = TRUE)) {
  ## LOG1S
  rasOrig <- terra::rast(ncols = 10, nrows = 10)
  ras <- rasOrig
  ras[] <- rep(c(0,1),50)
  assessDataType(ras)

  ras <- rasOrig
  ras[] <- rep(c(0,1),50)
  assessDataType(ras)

  ras[] <- rep(c(TRUE,FALSE),50)
  assessDataType(ras)

  ras[] <- c(NA, NA, rep(c(0,1),49))
  assessDataType(ras)

  ras <- rasOrig
  ras[] <- c(0, NaN, rep(c(0,1),49))
  assessDataType(ras)


  ## INT1S
  ras[] <- -1:98
  assessDataType(ras)

  ras[] <- c(NA, -1:97)
  assessDataType(ras)

  ## INT1U
  ras <- rasOrig
  ras[] <- 1:100
  assessDataType(ras)

  ras[] <- c(NA, 2:100)
  assessDataType(ras)

  ## INT2U
  ras <- rasOrig
  ras[] <- round(runif(100, min = 64000, max = 65000))
  assessDataType(ras)

  ## INT2S
  ras <- rasOrig
  ras[] <- round(runif(100, min = -32767, max = 32767))
  assessDataType(ras)

  ras[54] <- NA
  assessDataType(ras)

  ## INT4U
```

```
ras <- rasOrig
ras[] <- round(runif(100, min = 0, max = 500000000))
assessDataType(ras)

ras[14] <- NA
assessDataType(ras)

## INT4S
ras <- rasOrig
ras[] <- round(runif(100, min = -200000000, max = 200000000))
assessDataType(ras)

ras[14] <- NA
assessDataType(ras)

## FLT4S
ras <- rasOrig
ras[] <- runif(100, min = -10, max = 87)
assessDataType(ras)

ras <- rasOrig
ras[] <- round(runif(100, min = -3.4e+26, max = 3.4e+28))
assessDataType(ras)

ras <- rasOrig
ras[] <- round(runif(100, min = 3.4e+26, max = 3.4e+28))
assessDataType(ras)

ras <- rasOrig
ras[] <- round(runif(100, min = -3.4e+26, max = -1))
assessDataType(ras)

## FLT8S
ras <- rasOrig
ras[] <- c(-Inf, 1, rep(c(0,1),49))
assessDataType(ras)

ras <- rasOrig
ras[] <- c(Inf, 1, rep(c(0,1),49))
assessDataType(ras)

ras <- rasOrig
ras[] <- round(runif(100, min = -1.7e+30, max = 1.7e+308))
assessDataType(ras)

ras <- rasOrig
ras[] <- round(runif(100, min = 1.7e+30, max = 1.7e+308))
assessDataType(ras)

ras <- rasOrig
ras[] <- round(runif(100, min = -1.7e+308, max = -1))
assessDataType(ras)
```

```
  # 2 layer with different types LOG1S and FLT8S
  ras <- rasOrig
  ras[] <- rep(c(0,1),50)
  ras1 <- rasOrig
  ras1[] <- round(runif(100, min = -1.7e+308, max = -1))
  sta <- c(ras, ras1)
  assessDataType(sta)

}
```

---

basename2 *A version of* base::basename *that is* NULL *resistant*

Description

A version of base::basename that is NULL resistant

Usage

```
basename2(x)
```

Arguments

x	A character vector of paths

Value

NULL if x is NULL, otherwise, as basename.

Same as base::basename()

Examples

```
basename2(NULL)
basename2("/home/user/file.txt")
```

Cache *Saves a wide variety function call outputs to disk and optionally RAM, for recovery later*

Description

[Stable]

A function that can be used to wrap around other functions to cache function calls for later use. This is normally most effective when the function to cache is slow to run, yet the inputs and outputs are small. The benefit of caching, therefore, will decline when the computational time of the "first" function call is fast and/or the argument values and return objects are large. The default setting (and first call to Cache) will always save to disk. The 2nd call to the same function will return from disk, unless `options("reproducible.useMemoise" = TRUE)`, then the 2nd time will recover the object from RAM and is normally much faster (at the expense of RAM use).

Usage

```
Cache(
  FUN,
  ...,
  dryRun = getOption("reproducible.dryRun", FALSE),
  notOlderThan = NULL,
  .objects = NULL,
  .cacheExtra = NULL,
  .functionName = NULL,
  .cacheChaining = getOption("reproducible.cacheChaining", NULL),
  outputObjects = NULL,
  algo = "xxhash64",
  cachePath = NULL,
  length = getOption("reproducible.length", Inf),
  userTags = c(),
  omitArgs = NULL,
  classOptions = list(),
  debugCache = character(),
  quick = getOption("reproducible.quick", FALSE),
  verbose = getOption("reproducible.verbose", 1),
  cacheId = NULL,
  cacheSaveFormat = getOption("reproducible.cacheSaveFormat"),
  useCache = getOption("reproducible.useCache", TRUE),
  useCloud = getOption("reproducible.useCloud", FALSE),
  cloudFolderID = getOption("reproducible.cloudFolderID", NULL),
  showSimilar = getOption("reproducible.showSimilar", FALSE),
  drv = getOption("reproducible.drv", NULL),
  conn = getOption("reproducible.conn", NULL)
)

cache2(
  FUN,
  ...,
  dryRun = getOption("reproducible.dryRun", FALSE),
  notOlderThan = NULL,
  .objects = NULL,
  .cacheExtra = NULL,
```

```
  .functionName = NULL,
  .cacheChaining = getOption("reproducible.cacheChaining", NULL),
  outputObjects = NULL,
  algo = "xxhash64",
  cachePath = NULL,
  length = getOption("reproducible.length", Inf),
  userTags = c(),
  omitArgs = NULL,
  classOptions = list(),
  debugCache = character(),
  quick = getOption("reproducible.quick", FALSE),
  verbose = getOption("reproducible.verbose", 1),
  cacheId = NULL,
  cacheSaveFormat = getOption("reproducible.cacheSaveFormat"),
  useCache = getOption("reproducible.useCache", TRUE),
  useCloud = getOption("reproducible.useCloud", FALSE),
  cloudFolderID = getOption("reproducible.cloudFolderID", NULL),
  showSimilar = getOption("reproducible.showSimilar", FALSE),
  drv = getOption("reproducible.drv", NULL),
  conn = getOption("reproducible.conn", NULL)
)

CacheV2(
  FUN,
  ...,
  notOlderThan = NULL,
  .objects = NULL,
  .cacheExtra = NULL,
  .functionName = NULL,
  outputObjects = NULL,
  algo = "xxhash64",
  cacheRepo = NULL,
  cachePath = NULL,
  length = getOption("reproducible.length", Inf),
  compareRasterFileLength,
  userTags = c(),
  omitArgs = NULL,
  classOptions = list(),
  debugCache = character(),
  makeCopy = FALSE,
  quick = getOption("reproducible.quick", FALSE),
  verbose = getOption("reproducible.verbose", 1),
  cacheId = NULL,
  useCache = getOption("reproducible.useCache", TRUE),
  useCloud = FALSE,
  cloudFolderID = NULL,
  showSimilar = getOption("reproducible.showSimilar", FALSE),
  drv = getDrv(getOption("reproducible.drv", NULL)),
```

```
  conn = getOption("reproducible.conn", NULL)
)
```

Arguments

FUN	Either a function (e.g., rnorm), a function call (e.g., rnorm(1)), or an unevaluated function call (e.g., using quote()).
...	Arguments passed to FUN, if FUN is not an expression.
dryRun	See reproducibleOptions.
notOlderThan	A time. Load an object from the Cache if it was created after this.
.objects	Character vector of objects to be digested. This is only applicable if there is a list, environment (or similar) with named objects within it. Only this/these objects will be considered for caching, i.e., only use a subset of the list, environment or similar objects. In the case of nested list-type objects, this will only be applied outermost first.
.cacheExtra	A an arbitrary R object that will be included in the CacheDigest, but otherwise not passed into the FUN. If the user supplies a named list, then Cache will report which individual elements of .cacheExtra have changed when options("reproducible.showSimilar = TRUE). This can allow a user more control and understanding for debugging.
.functionName	A an arbitrary character string that provides a name that is different than the actual function name (e.g., "rnorm") which will be used for messaging. This can be useful when the actual function is not helpful for a user, such as do.call.
.cacheChaining	A logical or a the name of a function. If TRUE, then the current Cache call will evaluate the function "outside" the Cache call (via sys.function(-1)) and attach the digest of that outer function to the entry for this Cache call. This will then be used by any subsequent Cache call within the same function. If the outer function is unchanged, and there is one or more objects that had been returned by a previous Cache call, then those objects will not be digested; rather their cacheId tag will be used in place of a new digest. This *should* cause no change in Caching outcomes, and it should be faster in cases where there are several Cache calls within the same function. If FALSE (current default), then this feature is not used. If set to NULL (i.e., unset, the current default), then it will not use cache chaining, but it will attach more information to the Cache entries for each cacheId, as well as new entries for "surroundingFunction" digest, so that if a user switches to .cacheChaining = TRUE, then it will be able to begin using cache chaining without needing to rerun the calls again. Can be set by an option.
outputObjects	Optional character vector indicating which objects to return. This is only relevant for list, environment (or similar) objects
algo	The digest algorithm to use. Default xxhash64 (see digest::digest() for others).
cachePath	A repository used for storing cached objects. This is optional if Cache is used inside a SpaDES module.
length	Numeric. If the element passed to Cache is a Path class object (from e.g., asPath(filename)) or it is a Raster with file-backing, then this will be passed

to digest::digest, essentially limiting the number of bytes to digest (for speed). This will only be used if quick = FALSE. Default is getOption("reproducible.length"), which is set to Inf.

userTags — A character vector with descriptions of the Cache function call. These will be added to the Cache so that this entry in the Cache can be found using userTags e.g., via showCache().

omitArgs — Optional. A character vector of argument names in FUN to omit from the cache digest, or TRUE to omit *every* captured argument (the digest is then based on FUN itself – including its body, so a meaningful edit to the function source still busts the cache – and on .cacheExtra). Useful when the developer wants the cache to be insensitive to the function's inputs and pin freshness via .cacheExtra instead.

classOptions — Optional list. This will pass into .robustDigest for specific classes. Should be options that the .robustDigest knows what to do with.

debugCache — Character or Logical. Either "complete" or "quick" (uses partial matching, so "c" or "q" work). TRUE is equivalent to "complete". If "complete", then the returned object from the Cache function will have two attributes, debugCache1 and debugCache2, which are the entire list(...) and that same object, but after all .robustDigest calls, at the moment that it is digested using digest, respectively. This attr(mySimOut, "debugCache2") can then be compared to a subsequent call and individual items within the object attr(mySimOut, "debugCache1") can be compared. If "quick", then it will return the same two objects directly, without evalutating the FUN(...).

quick — Logical or character. If TRUE, no disk-based information will be assessed, i.e., only memory content. See Details section about quick in Cache().

verbose — Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

cacheId — Character string. If passed, this will override the calculated hash of the inputs, and return the result from this cacheId in the cachePath. Setting this is equivalent to manually saving the output of this function, i.e., the object will be on disk, and will be recovered in subsequent This may help in some particularly finicky situations where Cache is not correctly detecting unchanged inputs. This will guarantee the object will be identical each time; this may be useful in operational code.

cacheSaveFormat — Character string: currently either qs or rds. Defaults to getOption("reproducible.cacheSaveFormat" qs may be faster but appears to have narrower range of conditions that work; rds is safer, and may be slower.

useCache — Logical, numeric or "overwrite" or "devMode". See details.

useCloud — Logical (TRUE / FALSE / NULL) or one of "pull" / "push". See Details.

cloudFolderID — A googledrive dribble of a folder, e.g., using drive_mkdir(). If left as NULL, the function will create a cloud folder with name from last two folder levels of the cachePath path, : paste0(basename(dirname(cachePath)), "_",

	basename(cachePath)). This cloudFolderID will be added to options("reproducible.cloudFolder but this will not persist across sessions. If this is a character string, it will treat this as a folder name to create or use on GoogleDrive.
showSimilar	A logical or numeric. Useful for debugging. If TRUE or 1, then if the Cache does not find an identical archive in the cachePath, it will report (via message) the next most recent similar archive, and indicate which argument(s) is/are different. If a number larger than 1, then it will report the N most recent similar archived objects.
drv	If using a database backend, drv must be an object that inherits from DBIDriver (e.g., RSQLite::SQLite).
conn	an optional DBIConnection object, as returned by dbConnect().
cacheRepo	Same as cachePath, but kept for backwards compatibility.
compareRasterFileLength	Being deprecated; use length.
makeCopy	Now deprecated. Ignored if used.

Details

There are other similar functions in the R universe. This version of Cache has been used as part of a robust continuous workflow approach. As a result, we have tested it with many "non-standard" R objects (e.g., RasterLayer, Spat* objects) and environments (which are always unique, so do not cache readily).

This version of the Cache function accommodates those four special, though quite common, cases by:

1. converting any environments into list equivalents;
2. identifying the dispatched S4 method (including those made through inheritance) before hashing so the correct method is being cached;
3. by hashing the linked file, rather than the raster object. Currently, only file-backed Raster* or Spat* objects are digested (e.g., not ff objects, or any other R object where the data are on disk instead of in RAM);
4. Uses digest::digest() This is used for file-backed objects as well.
5. Cache will save arguments passed by user in a hidden environment. Any nested Cache functions will use arguments in this order: 1) actual arguments passed at each Cache call; 2) any inherited arguments from an outer Cache call; 3) the default values of the Cache function. See section on *Nested Caching*.

The algorithm used to compute the cacheId (the hash of the inputs) is selectable via the reproducible.digestVersion option; see the digestVersion entry in reproducibleOptions() for the available versions and what each changes (notably version 4, which makes sf/SpatVector digests identical across operating systems).

Cache will add a tag to the entry in the cache database called accessed, which will assign the time that it was accessed, either read or write. That way, cached items can be shown (using showCache) or removed (using clearCache) selectively, based on their access dates, rather than only by their creation dates. See example in clearCache().

Value

Returns the value of the function call or the cached version (i.e., the result from a previous call to this same cached function with identical arguments).

Nested Caching

Commonly, Caching is nested, i.e., an outer function is wrapped in a `Cache` function call, and one or more inner functions are also wrapped in a `Cache` function call. A user *can* always specify arguments in every Cache function call, but this can get tedious and can be prone to errors. The normal way that R handles arguments is it takes the user passed arguments if any, and default arguments for all those that have no user passed arguments. We have inserted a middle step. The order or precedence for any given `Cache` function call is

1. user arguments, 2. inherited arguments, 3. default arguments. At this time, the top level `Cache` arguments will propagate to all inner functions unless each individual `Cache` call has other arguments specified, i.e., "middle" nested `Cache` function calls don't propagate their arguments to further "inner" `Cache` function calls. See example.

`userTags` is unique of all arguments: its values will be appended to the inherited `userTags`.

quick

The `quick` argument is attempting to sort out an ambiguity with character strings: are they file paths or are they simply character strings. When `quick = TRUE`, `Cache` will treat these as character strings; when `quick = FALSE`, they will be attempted to be treated as file paths first; if there is no file, then it will revert to treating them as character strings. If user passes a character vector to this, then this will behave like `omitArgs`: `quick = "file"` will treat the argument `"file"` as character string.

The most often encountered situation where this ambiguity matters is in arguments about filenames: is the filename an input pointing to an object whose content we want to assess (e.g., a file-backed raster), or an output (as in saveRDS) and it should not be assessed. If only run once, the output file won't exist, so it will be treated as a character string. However, once the function has been run once, the output file will exist, and `Cache(...)` will assess it, which is incorrect. In these cases, the user is advised to use `quick = "TheOutputFilenameArgument"` to specify the argument whose content on disk should not be assessed, but whose character string should be assessed (distinguishing it from `omitArgs = "TheOutputFilenameArgument"`, which will not assess the file content nor the character string).

This is relevant for objects of class `character`, `Path` and `Raster` currently. For class `character`, it is ambiguous whether this represents a character string or a vector of file paths. If it is known that character strings should not be treated as paths, then `quick = TRUE` is appropriate, with no loss of information. If it is file or directory, then it will digest the file content, or `basename(object)`. For class `Path` objects, the file's metadata (i.e., filename and file size) will be hashed instead of the file contents if `quick = TRUE`. If set to `FALSE` (default), the contents of the file(s) are hashed. If `quick = TRUE`, `length` is ignored. `Raster` objects are treated as paths, if they are file-backed.

Caching Speed

Caching speed may become a critical aspect of a final product. For example, if the final product is a shiny app, rerunning the entire project may need to take less then a few seconds at most. There are 3

arguments that affect Cache speed: quick, length, and algo. quick is passed to .robustDigest, which currently only affects Path and Raster* class objects. In both cases, quick means that little or no disk-based information will be assessed.

Filepaths

If a function has a path argument, there is some ambiguity about what should be done. Possibilities include:

1. hash the string as is (this will be very system specific, meaning a Cache call will not work if copied between systems or directories);
2. hash the basename(path);
3. hash the contents of the file.

If paths are passed in as is (i.e,. character string), the result will not be predictable. Instead, one should use the wrapper function asPath(path), which sets the class of the string to a Path, and one should decide whether one wants to digest the content of the file (using quick = FALSE), or just the filename ((quick = TRUE)). See examples.

Stochasticity or randomness

In general, it is expected that caching will only be used when randomness is not desired, e.g., Cache(rnorm(1)) is unlikely to be useful in many cases. However, Cache captures the call that is passed to it, leaving all functions unevaluated. As a result Cache(glm, x ~ y, rnorm(1)) will not work as a means of forcing a new evaluation each time, as the rnorm(1) is not evaluated before the call is assessed against the cache database. To force a new call each time, evaluate the randomness prior to the Cache call, e.g., ran = rnorm(1) then pass this to .cacheExtra, e.g., Cache(glm, x ~ y, .cacheExtra = ran)

drv and conn

By default, drv uses an SQLite database. This can be sufficient for most cases. However, if a user has dozens or more cores making requests to the Cache database, it may be insufficient. A user can set up a different database backend, e.g., PostgreSQL that can handle multiple simultaneous read-write situations. See https://github.com/PredictiveEcology/SpaDES/wiki/Using-alternate-database-backends

useCache

Logical or numeric. If FALSE or 0, then the entire Caching mechanism is bypassed and the function is evaluated as if it was not being Cached. Default is getOption("reproducible.useCache")), which is TRUE by default, meaning use the Cache mechanism. This may be useful to turn all Caching on or off in very complex scripts and nested functions. Increasing levels of numeric values will cause deeper levels of Caching to occur (though this may not work as expected in all cases). The following is no longer supported: Currently, only implemented in postProcess: to do both caching of inner cropInputs, projectInputs and maskInputs, and caching of outer postProcess, use useCache = 2; to skip the inner sequence of 3 functions, use useCache = 1. For large objects, this may prevent many duplicated save to disk events.

If useCache = "overwrite" (which can be set with options("reproducible.useCache" = "overwrite")), then the function invoke the caching mechanism but will purge any entry that is matched, and it will be replaced with the results of the current call.

If useCache = "devMode": The point of this mode is to facilitate using the Cache when functions and datasets are continually in flux, and old Cache entries are likely stale very often. In devMode, the cache mechanism will work as normal if the Cache call is the first time for a function OR if it successfully finds a copy in the cache based on the normal Cache mechanism. It *differs* from the normal Cache if the Cache call does *not* find a copy in the cachePath, but it does find an entry that matches based on userTags. In this case, it will delete the old entry in the cachePath (identified based on matching userTags), then continue with normal Cache. For this to work correctly, userTags must be unique for each function call. This should be used with caution as it is still experimental. Currently, if userTags are not unique to a single entry in the cachePath, it will default to the behaviour of useCache = TRUE with a message. This means that "devMode" is most useful if used from the start of a project.

useCloud

This is experimental and there are many conditions under which this is known to not work correctly. This is a way to store all or some of the local Cache in the cloud. Currently, the only cloud option is Google Drive, via **googledrive**. For this to work, the user must be or be able to be authenticated with googledrive::drive_auth. The principle behind this useCloud is that it will be a full or partial mirror of a local Cache. It is not intended to be used independently from a local Cache. To share objects that are in the Cloud with another person, it requires 2 steps. 1) share the cloudFolderID$id, which can be retrieved by getOption("reproducible.cloudFolderID")$id after at least one Cache call has been made. 2) The other user must then set their cacheFolderID in a Cache\(..., reproducible.cloudFol call or set their option manually options\(\"reproducible.cloudFolderID\" = \"the ID here\"\).

If TRUE, then this Cache call will download (if local copy doesn't exist, but cloud copy does exist), upload (local copy does or doesn't exist and cloud copy doesn't exist), or will not download nor upload if object exists in both. If TRUE will be at least 1 second slower than setting this to FALSE, and likely even slower as the cloud folder gets large. If a user wishes to keep "high-level" control, set this to getOption("reproducible.useCloud", FALSE) or getOption("reproducible.useCloud", TRUE) (if the default behaviour should be FALSE or TRUE, respectively) so it can be turned on and off with this option. NOTE: *This argument will not be passed into inner/nested Cache calls.*)

Two character values are also accepted, intended for separating developer and user roles when sharing a cloud-cache folder:

- "push" is equivalent to TRUE (developer role) – bidirectional; downloads on a cloud hit, uploads on a miss.
- "pull" is read-only (user role) – downloads on a cloud hit, but never uploads. If the local cache already has the object, the cloud is not consulted at all (the Google Drive listing is deferred until after the local lookup fails). When neither local nor cloud has the object, the call falls back to a normal local-only Cache run.

Object attributes

Users should be cautioned that object attributes may not be preserved, especially in the case of objects that are file-backed, such as Raster or SpatRaster objects. If a user needs to keep attributes, they may need to manually re-attach them to the object after recovery. With the example of SpatRaster objects, saving to disk requires terra::wrap if it is a memory-backed object. When running terra::unwrap on this object, any attributes that a user had added are lost.

`sideEffect`

This feature is now deprecated. Do not use as it is ignored.

Note

As indicated above, several objects require pre-treatment before caching will work as expected. The function `.robustDigest` accommodates this. It is an S4 generic, meaning that developers can produce their own methods for different classes of objects. Currently, there are methods for several types of classes. See `.robustDigest()`.

Author(s)

Eliot McIntire

See Also

`showCache()`, `clearCache()`, `keepCache()`, `CacheDigest()` to determine the digest of a given function or expression, as used internally within Cache, `movedCache()`, `.robustDigest()`, `reproducibleOptions()` (e.g. the `digestVersion` option that selects the `cacheId` algorithm), and for more advanced uses there are several helper functions, e.g., `rmFromCache()`, `CacheStorageDir()`

Examples

```
data.table::setDTthreads(2)
tmpDir <- tempdir()
opts <- options(reproducible.cachePath = tmpDir)

# Usage -- All below are equivalent; even where args are missing or provided,
#   Cache evaluates using default values, if these are specified in formals(FUN)
a <- list()
b <- list(fun = rnorm)
bbb <- 1
ee <- new.env(parent = emptyenv())
ee$qq <- bbb

a[[1]] <- Cache(rnorm(1)) # no evaluation prior to Cache
a[[2]] <- Cache(rnorm, 1) # no evaluation prior to Cache
a[[3]] <- Cache(do.call, rnorm, list(1))
a[[4]] <- Cache(do.call(rnorm, list(1)))
a[[5]] <- Cache(do.call(b$fun, list(1)))
a[[6]] <- Cache(do.call, b$fun, list(1))
a[[7]] <- Cache(b$fun, 1)
a[[8]] <- Cache(b$fun(1))
a[[10]] <- Cache(quote(rnorm(1)))
a[[11]] <- Cache(stats::rnorm(1))
a[[12]] <- Cache(stats::rnorm, 1)
a[[13]] <- Cache(rnorm(1, 0, get("bbb", inherits = FALSE)))
a[[14]] <- Cache(rnorm(1, 0, get("qq", inherits = FALSE, envir = ee)))
a[[15]] <- Cache(rnorm(1, bbb - bbb, get("bbb", inherits = FALSE)))
a[[16]] <- Cache(rnorm(sd = 1, 0, n = get("bbb", inherits = FALSE))) # change order
a[[17]] <- Cache(rnorm(1, sd = get("ee", inherits = FALSE)$qq), mean = 0)
```

```
# with base pipe -- this is put in quotes ('') because R version 4.0 can't understand this
#  if you are using R >= 4.1 or R >= 4.2 if using the _ placeholder,
#  then you can just use pipe normally
usingPipe1 <- "b$fun(1) |> Cache()"  # base pipe

# For long pipe, need to wrap sequence in { }, or else only last step is cached
usingPipe2 <-
  '{"bbb" |>
      parse(text = _) |>
      eval() |>
      rnorm()} |>
    Cache()'
a[[9]] <- eval(parse(text = usingPipe1)) # recovers cached copy
a[[18]] <- eval(parse(text = usingPipe2)) # recovers cached copy

length(unique(a)) == 1 #  all same

### Pipe -- have to use { } or else only final function is Cached
b1a <- 'sample(1e5, 1) |> rnorm() |> Cache()'
b1b <- 'sample(1e5, 1) |> rnorm() |> Cache()'
b2a <- '{sample(1e5, 1) |> rnorm()} |> Cache()'
b2b <- '{sample(1e5, 1) |> rnorm()} |> Cache()'
b1a <- eval(parse(text = b1a))
b1b <- eval(parse(text = b1b))
b2a <- eval(parse(text = b2a))
b2b <- eval(parse(text = b2b))
all.equal(b1a, b1b) # Not TRUE because the sample is run first
all.equal(b2a, b2b) # TRUE because of {  }, sample is not run

##########################
# Advanced examples
##########################

# .cacheExtra -- add something to digest
Cache(rnorm(1), .cacheExtra = "sfessee11") # adds something other than fn args
Cache(rnorm(1), .cacheExtra = "nothing") # even though fn is same, the extra is different

# omitArgs -- remove something from digest (kind of the opposite of .cacheExtra)
Cache(rnorm(2, sd = 1), omitArgs = "sd") # removes one or more args from cache digest
Cache(rnorm(2, sd = 2), omitArgs = "sd") # b/c sd is not used, this is same as previous

# cacheId -- force the use of a digest -- can give undesired consequences
Cache(rnorm(3), cacheId = "k323431232") # sets the cacheId for this call
Cache(runif(14), cacheId = "k323431232") # recovers same as above, i.e, rnorm(3)

# Turn off Caching session-wide
opts <- options(reproducible.useCache = FALSE)
Cache(rnorm(3)) # doesn't cache
options(opts)

# showSimilar can help with debugging why a Cache call isn't picking up a cached copy
Cache(rnorm(4), showSimilar = TRUE) # shows that the argument `n` is different
```

```
##############################################
# devMode -- enables cache database to stay
#            small even when developing code
##############################################
opt <- options("reproducible.useCache" = "devMode")
clearCache(tmpDir, ask = FALSE)
centralTendency <- function(x) {
  mean(x)
}
funnyData <- c(1, 1, 1, 1, 10)
uniqueUserTags <- c("thisIsUnique", "reallyUnique")
ranNumsB <- Cache(centralTendency, funnyData, cachePath = tmpDir,
                  userTags = uniqueUserTags) # sets new value to Cache
showCache(tmpDir) # 1 unique cacheId -- cacheId is 71cd24ec3b0d0cac

# During development, we often redefine function internals
centralTendency <- function(x) {
  median(x)
}
# When we rerun, we don't want to keep the "old" cache because the function will
#   never again be defined that way. Here, because of userTags being the same,
#   it will replace the entry in the Cache, effetively overwriting it, even though
#   it has a different cacheId
ranNumsD <- Cache(centralTendency, funnyData, cachePath = tmpDir, userTags = uniqueUserTags)
showCache(tmpDir) # 1 unique artifact -- cacheId is 632cd06f30e111be

# If it finds it by cacheID, doesn't matter what the userTags are
ranNumsD <- Cache(centralTendency, funnyData, cachePath = tmpDir, userTags = "thisIsUnique")
options(opt)

########################################
# For more in depth uses, see vignette
if (interactive())
  browseVignettes(package = "reproducible")
```

`CacheDigest` *The exact digest function that* `Cache` *uses*

Description

This can be used by a user to pre-test their arguments before running `Cache`, for example to determine whether there is a cached copy.

Usage

```
CacheDigest(
  objsToDigest,
  ...,
  algo = "xxhash64",
```

```
  calledFrom = "CacheDigest",
  .functionName = NULL,
  quick = FALSE
)
```

Arguments

objsToDigest	A list of all the objects (e.g., arguments) to be digested
...	passed to .robustDigest.
algo	The digest algorithm to use. Default xxhash64 (see digest::digest() for others).
calledFrom	a Character string, length 1, with the function to compare with. Default is "Cache". All other values may not produce robust CacheDigest results.
.functionName	A an arbitrary character string that provides a name that is different than the actual function name (e.g., "rnorm") which will be used for messaging. This can be useful when the actual function is not helpful for a user, such as do.call.
quick	Logical or character. If TRUE, no disk-based information will be assessed, i.e., only memory content. See Details section about quick in Cache().

Value

A list of length 2 with the outputHash, which is the digest that Cache uses for cacheId and also preDigest, which is the digest of each sub-element in objsToDigest.

Examples

```
data.table::setDTthreads(2)
a <- Cache(rnorm, 1)

# like with Cache, user can pass function and args in a few ways
CacheDigest(rnorm(1)) # shows same cacheId as previous line
CacheDigest(rnorm, 1) # shows same cacheId as previous line
```

CacheGeo *Cache-like function for spatial domains*

Description

[Experimental]

This function is a combination of Cache and prepInputs but for spatial domains. This differs from Cache in that the current function call doesn't have to have an identical function call previously run. Instead, it needs to have had a previous function call where the domain being passes is *within* the geographic limits of the targetFile. This is similar to a geospatial operation on a remote GIS server, with 2 differences:

1. This downloads the object first before doing the GIS locally, and 2. it will optionally upload an updated object if the geographic area did not yet exist.

Usage

```
CacheGeo(
  targetFile = NULL,
  domain,
  FUN,
  destinationPath = getOption("reproducible.destinationPath", "."),
  useCloud = getOption("reproducible.useCloud", FALSE),
  cloudFolderID = NULL,
  purge = FALSE,
  useCache = getOption("reproducible.useCache"),
  overwrite = getOption("reproducible.overwrite"),
  action = c("nothing", "update", "replace", "append"),
  bufferOK = FALSE,
  verbose = getOption("reproducible.verbose"),
  ...
)
```

Arguments

targetFile	The (optional) local file (or path to file) name for a sf object or data.frame that can be coerced to a sf object (i.e., has a geometry column). If cloudFolderID is specified, then this will be the name of the file stored and/or accessed in that cloud folder.
domain	An sf polygon object that is the spatial area of interest. If NULL, then this will return the whole object in targetFile.
FUN	A function call that will be called if there is the domain is not already contained within the sf object at targetFile. This function call MUST return either a sf class object or a data.frame class object that has a geometry column (which can then be converted to sf with sf::st_as_sf)
destinationPath	Character string of a directory in which to download and save the file that comes from url and is also where the function will look for archive or targetFile. To prevent repeated downloads in different locations, the user can also set options("reproducible.dest to one or more local file paths to search for the file before attempting to download. Default for that option is NULL meaning do not search locally. The previous name options("reproducible.inputPaths") is still accepted as a backwards-compatible alias.
useCloud	A logical.
cloudFolderID	If this is specified, then it must be either 1) a Google Drive url to a folder where the targetFile will be read from or written to, or 2) a googledrive id or 3) an absolute path to a (possibly non-existent yet) folder on your Google drive.
purge	Logical or Integer. 0/FALSE (default) keeps existing CHECKSUMS.txt file and prepInputs will write or append to it. 1/TRUE will deleted the entire CHECKSUMS.txt file. Other options, see details.
useCache	Passed to Cache in various places. Defaults to getOption("reproducible.useCache", 2L) in prepInputs, and getOption("reproducible.useCache", FALSE) if

calling any of the inner functions manually. For prepInputs, this mean it will use Cache only up to 2 nested levels, which includes preProcess. postProcess and its nested *Input functions (e.g., cropInputs, projectInputs, maskInputs) are no longer internally cached, as terra processing speeds mean internal caching is more time consuming. We recommend caching the full prepInputs call instead (e.g. prepInputs(...) |> Cache()).

overwrite — Logical. Passed to writeTo (possibly inside postProcess) and postProcess.

action — A character string, with one of c("nothing", "update", "replace", "append"). Partial matching is used ("n" is sufficient). nothing will prevent any updating of the targetFile, i.e., "read only". append will add the spatial elements in domain to targetFile (and writing it back to disk). update will do the same as append, but will also remove any identical geometries before appending. replace does nothing currently.

bufferOK — A logical. If TRUE, then after testing whether the domain is within the targetFile spatial object, and if it returns FALSE, then the function will create a larger object, buffered by 2.5% of the extent of the object. If FALSE, then it will be strict about whether the domain is within the targetFile.

verbose — Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

... — All named objects that are needed for FUN, including the function itself, if it is not in a package.

Details

This has a very specific use case: assess whether an existing sf polygon or multipolygon object (local or remote) covers the spatial area of a domain of interest. If it does, then return only that part of the sf object that completely covers the domain. If it does not, then run FUN. It is expected that FUN will produce an sf polygon or multipolygon class object. The result of FUN will then be appended to the sf object as a new entry (feature) or it will replace the existing "same extent" entry in the sf object.

Value

Returns an object that results from FUN, which will possibly be a subset of a larger spatial object that is specified with targetFile.

Examples

```
if (requireNamespace("sf", quietly = TRUE) &&
    requireNamespace("terra", quietly = TRUE)) {
  dPath <- checkPath(file.path(tempdir2()), create = TRUE)
  localFileLux <- system.file("ex/lux.shp", package = "terra")

  # 1 step for each layer
  # 1st step -- get study area
  full <- prepInputs(localFileLux, destinationPath = dPath) # default is sf::st_read
```

```
  zoneA <- full[3:6, ]
  zoneB <- full[8, ] # not in A
  zoneC <- full[3, ] # yes in A
  zoneD <- full[7:8, ] # not in A, B or C
  zoneE <- full[3:5, ] # yes in A
  # 2nd step: re-write to disk as read/write is lossy; want all "from disk" for this ex.
  writeTo(zoneA, writeTo = "zoneA.shp", destinationPath = dPath)
  writeTo(zoneB, writeTo = "zoneB.shp", destinationPath = dPath)
  writeTo(zoneC, writeTo = "zoneC.shp", destinationPath = dPath)
  writeTo(zoneD, writeTo = "zoneD.shp", destinationPath = dPath)
  writeTo(zoneE, writeTo = "zoneE.shp", destinationPath = dPath)
  # Must re-read to get identical columns
  zoneA <- sf::st_read(file.path(dPath, "zoneA.shp"))
  zoneB <- sf::st_read(file.path(dPath, "zoneB.shp"))
  zoneC <- sf::st_read(file.path(dPath, "zoneC.shp"))
  zoneD <- sf::st_read(file.path(dPath, "zoneD.shp"))
  zoneE <- sf::st_read(file.path(dPath, "zoneE.shp"))

  # The function that is to be run. This example returns a data.frame because
  #    saving `sf` class objects with list-like columns does not work with
  #    many st_driver()
  fun <- function(domain, newField) {
    domain |>
      as.data.frame() |>
      cbind(params = I(lapply(seq_len(NROW(domain)), function(x) newField)))
  }

  # Run sequence -- A, B will add new entries in targetFile, C will not,
  #                 D will, E will not
  for (z in list(zoneA, zoneB, zoneC, zoneD, zoneE)) {
    out <- CacheGeo(
      targetFile = "fireSenseParams.rds",
      domain = z,
      FUN = fun(domain, newField = I(list(list(a = 1, b = 1:2, c = "D")))),
      fun = fun, # pass whatever is needed into the function
      destinationPath = dPath,
      action = "update"
      # , cloudFolderID = "cachedObjects" # to upload/download from cloud
    )
  }
}
```

`cacheId` *Extract the cache id of an object*

Description

Any object that was returned from the Cache or was calculated as part of a Cache call will have an attribute, `tags` and an entry with `cacheId:` prefix. This is a lightweight helper to extract that `cacheId`.

Usage

```
cacheId(obj)
```

Arguments

obj	Any R object

Value

The cacheId if this was part of a Cache call. Otherwise NULL

Examples

```
# An object that was never Cached has no cacheId
cacheId(1:10)
```

---

checkPath *Check directory path*

Description

Checks the specified path to a directory for formatting consistencies, such as trailing slashes, etc.

Usage

```
checkPath(path, create)

## S4 method for signature 'character,logical'
checkPath(path, create)

## S4 method for signature 'character,missing'
checkPath(path)

## S4 method for signature 'NULL,ANY'
checkPath(path)

## S4 method for signature 'missing,ANY'
checkPath()
```

Arguments

path	A character string corresponding to a directory path.
create	A logical indicating whether the path should be created if it does not exist. Default is FALSE.

Value

Character string denoting the cleaned up filepath.

Note

This will not work for paths to files. To check for existence of files, use `file.exists()`. To normalize a path to a file, use `normPath()` or `normalizePath()`.

See Also

`file.exists()`, `dir.create()`, `normPath()`

Examples

```
## normalize file paths
paths <- list("./aaa/zzz",
              "./aaa/zzz/",
              ".//aaa//zzz",
              ".//aaa//zzz/",
              ".\\\\aaa\\\\zzz",
              ".\\\\aaa\\\\zzz\\\\",
              file.path(".", "aaa", "zzz"))

checked <- normPath(paths)
length(unique(checked)) ## 1; all of the above are equivalent

## check to see if a path exists
tmpdir <- file.path(tempdir(), "example_checkPath")

dir.exists(tmpdir) ## FALSE
tryCatch(checkPath(tmpdir, create = FALSE), error = function(e) FALSE) ## FALSE

checkPath(tmpdir, create = TRUE)
dir.exists(tmpdir) ## TRUE

unlink(tmpdir, recursive = TRUE)
```

---

checkRelative *An alternative to* basename *and* dirname *when there are sub-folders*

Description

This confirms that the `files` which may be absolute actually exist when compared `makeRelative(knownRelativeFiles, absolutePrefix)`. This is different than just using `basename` because it will include any sub-folder structure within the `knownRelativePaths`

Usage

```
checkRelative(
  files,
  absolutePrefix,
  knownRelativeFiles,
  verbose = getOption("reproducible.verbose")
)
```

Arguments

files	A character vector of files to check to see if they are the same as knownRelativeFiles, once the absolutePrefix is removed
absolutePrefix	A directory to "remove" from files to compare to knownRelativeFiles
knownRelativeFiles	A character vector of relative filenames, that could have sub-folder structure.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

---

Checksums *Calculate checksum*

Description

Verify (and optionally write) checksums. Checksums are computed using .digest(), which is simply a wrapper around digest::digest.

Usage

```
Checksums(
  path,
  write,
  quickCheck = getOption("reproducible.quickCheck", FALSE),
  checksumFile = identifyCHECKSUMStxtFile(path),
  files = NULL,
  verbose = getOption("reproducible.verbose", 1),
  ...
)

## S4 method for signature 'character,logical'
Checksums(
  path,
  write,
  quickCheck = getOption("reproducible.quickCheck", FALSE),
  checksumFile = identifyCHECKSUMStxtFile(path),
```

```
  files = NULL,
  verbose = getOption("reproducible.verbose", 1),
  ...
)

## S4 method for signature 'character,missing'
Checksums(
  path,
  write,
  quickCheck = getOption("reproducible.quickCheck", FALSE),
  checksumFile = identifyCHECKSUMStxtFile(path),
  files = NULL,
  verbose = getOption("reproducible.verbose", 1),
  ...
)
```

Arguments

path	Character string giving the directory path containing CHECKSUMS.txt file, or where it will be written if checksumFile = TRUE.
write	Logical indicating whether to overwrite CHECKSUMS.txt. Default is FALSE, as users should not change this file. Module developers should write this file prior to distributing their module code, and update accordingly when the data change.
quickCheck	Logical. If TRUE, then this will only use file sizes, rather than a digest::digest hash. This is generally faster, but will be *much* less robust.
checksumFile	The filename of the checksums file to read or write to. The default is 'CHECKSUMS.txt' located at file.path(path, module, "data", checksumFile). It is likely not a good idea to change this, and should only be used in cases such as Cache, which can evaluate if the checksumFile has changed.
files	An optional character string or vector of specific files to checksum. This may be very important if there are many files listed in a CHECKSUMS.txt file, but only a few are to be checksummed.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
...	Passed to digest::digest() and utils::write.table(). For digest, the notable argument is algo. For write.table, the notable argument is append.

Value

A data.table with columns: result, expectedFile, actualFile, checksum.x, checksum.y, algorithm.x, algorithm.y, filesize.x, filesize.y indicating the result of comparison between local file (x) and expectation based on the CHECKSUMS.txt file.

Note

In version 1.2.0 and earlier, two checksums per file were required because of differences in the checksum hash values on Windows and Unix-like platforms. Recent versions use a different (faster) algorithm and only require one checksum value per file. To update your 'CHECKSUMS.txt' files using the new algorithm, see https://github.com/PredictiveEcology/SpaDES/issues/295#issuecomment-246513405.

Author(s)

Alex Chubaty

Examples

```
modulePath <- file.path(tempdir(), "myModulePath")
dir.create(modulePath, recursive = TRUE, showWarnings = FALSE)
moduleName <- "myModule"
cat("hi", file = file.path(modulePath, moduleName)) # put something there for this example

## verify checksums of all data files
Checksums(modulePath, files = moduleName)

## write new CHECKSUMS.txt file
Checksums(files = moduleName, modulePath, write = TRUE)
```

---

cloudDownload *Download from cloud, if necessary*

Description

Meant for internal use, as there are internal objects as arguments.

Usage

```
cloudDownload(
  outputHash,
  newFileName,
  gdriveLs,
  cachePath,
  cloudFolderID,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose")
)
```

Arguments

outputHash	The cacheId of the object to upload
newFileName	The character string of the local filename that the downloaded object will have
gdriveLs	The result of googledrive::drive_ls(googledrive::as_id(cloudFolderID), pattern = "outputHash")
cachePath	A repository used for storing cached objects. This is optional if Cache is used inside a SpaDES module.
cloudFolderID	A googledrive dribble of a folder, e.g., using drive_mkdir(). If left as NULL, the function will create a cloud folder with name from last two folder levels of the cachePath path, : paste0(basename(dirname(cachePath)), "_", basename(cachePath)). This cloudFolderID will be added to options("reproducible.cloudFolder but this will not persist across sessions. If this is a character string, it will treat this as a folder name to create or use on GoogleDrive.
drv	If using a database backend, drv must be an object that inherits from DBIDriver (e.g., RSQLite::SQLite).
conn	an optional DBIConnection object, as returned by dbConnect().
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

---

compareNA *NA-aware comparison of two vectors*

Description

Copied from http://www.cookbook-r.com/Manipulating_data/Comparing_vectors_or_factors_with_NA/. This function returns TRUE wherever elements are the same, including NA's, and FALSE everywhere else.

Usage

```
compareNA(v1, v2)
```

Arguments

v1	A vector
v2	A vector

Value

A logical vector, indicating positions where two vectors are same or differ.

Examples

```
a <- c(NA, 1, 2, NA)
b <- c(1, NA, 2, NA)
compareNA(a, b)
```

`convertCallToCommonFormat` *Convert all ways of calling a function into canonical form, including defaults*

Description

e.g., stats::rnorm(1) –> rnorm(n = 1, mean = 0, sd = 1)

Usage

```
convertCallToCommonFormat(call, usesDots, isSquiggly, .callingEnv)
```

Arguments

`call`	The full captured call as it was passed by user.
`usesDots`	Logical. Whether the original `Cache` call used `...`
`isSquiggly`	Logical. Whether there are curly braces e.g., as in a pipe sequence.
`.callingEnv`	Environment. The environment from which `Cache` was called.

`convertPaths` *Change the absolute path of a file*

Description

`convertPaths` is simply a wrapper around `gsub` for changing the first part of a path. `convertRasterPaths` is useful for changing the path to a file-backed raster (e.g., after copying the file to a new location).

Usage

```
convertPaths(x, patterns, replacements)

convertRasterPaths(x, patterns, replacements)
```

Arguments

x	For convertPaths, a character vector of file paths. For convertRasterPaths, a disk-backed RasterLayer object, or a list of such rasters.
patterns	Character vector containing a pattern to match (see ?gsub).
replacements	Character vector of the same length of patterns containing replacement text (see ?gsub).

Value

A normalized path with the patterns replaced by replacements. Or a list of such objects if x was a list.

Author(s)

Eliot McIntire and Alex Chubaty

Examples

```
filenames <- c("/home/user1/Documents/file.txt", "/Users/user1/Documents/file.txt")
oldPaths <- dirname(filenames)
newPaths <- c("/home/user2/Desktop", "/Users/user2/Desktop")
convertPaths(filenames, oldPaths, newPaths)
```

Copy *Recursive copying of nested environments, and other "hard to copy" objects*

Description

When copying environments and all the objects contained within them, there are no copies made: it is a pass-by-reference operation. Sometimes, a deep copy is needed, and sometimes, this must be recursive (i.e., environments inside environments).

Usage

```
Copy(object, ...)

## S4 method for signature 'ANY'
Copy(
  object,
  filebackedDir,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  ...
)
```

```
## S4 method for signature 'data.table'
Copy(object, ...)

## S4 method for signature 'list'
Copy(object, ...)

## S4 method for signature 'refClass'
Copy(object, ...)

## S4 method for signature 'data.frame'
Copy(object, ...)
```

Arguments

object	An R object (likely containing environments) or an environment.
...	Only used for custom Methods
filebackedDir	A directory to copy any files that are backing R objects, currently only valid for Raster classes. Defaults to .reproducibleTempPath(), which is unlikely to be very useful. Can be NULL, which means that the file will not be copied and could therefore cause a collision as the pre-copied object and post-copied object would have the same file backing them.
drv	If using a database backend, drv must be an object that inherits from DBIDriver (e.g., RSQLite::SQLite).
conn	an optional DBIConnection object, as returned by dbConnect().
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

Details

To create a new Copy method for a class that needs its own method, try something like shown in example and put it in your package (or other R structure).

Value

The same object as object, but with pass-by-reference class elements "deep" copied. reproducible has methods for several classes.

Author(s)

Eliot McIntire

See Also

.robustDigest(), Filenames()

Examples

```
e <- new.env()
e$abc <- letters
e$one <- 1L
e$lst <- list(W = 1:10, X = runif(10), Y = rnorm(10), Z = LETTERS[1:10])
ls(e)

# 'normal' copy
f <- e
ls(f)
f$one
f$one <- 2L
f$one
e$one ## uh oh, e has changed!

# deep copy
e$one <- 1L
g <- Copy(e)
ls(g)
g$one
g$one <- 3L
g$one
f$one
e$one
## To create a new deep copy method, use the following template
## setMethod("Copy", signature = "the class", # where = specify here if not in a package,
##           definition = function(object, filebackendDir, ...) {
##           # write deep copy code here
##           })
```

`copySingleFile` *Copy a file using* `robocopy` *on Windows and* `rsync` *on Linux/macOS*

Description

This is replacement for `file.copy`, but for one file at a time. The additional feature is that it will use `robocopy` (on Windows) or `rsync` on Linux or Mac, if they exist. It will default back to `file.copy` if none of these exists. If there is a possibility that the file already exists, then this function should be very fast as it will do "update only", i.e., nothing.

Usage

```
copySingleFile(
  from = NULL,
  to = NULL,
  useRobocopy = TRUE,
  overwrite = TRUE,
  delDestination = FALSE,
```

```
  create = TRUE,
  silent = FALSE
)

copyFile(
  from = NULL,
  to = NULL,
  useRobocopy = TRUE,
  overwrite = TRUE,
  delDestination = FALSE,
  create = TRUE,
  silent = FALSE
)
```

Arguments

from	The source file.
to	The new file.
useRobocopy	For Windows, this will use a system call to robocopy which appears to be much faster than the internal file.copy function. Uses /MIR flag. Default TRUE.
overwrite	Passed to file.copy
delDestination	Logical, whether the destination should have any files deleted, if they don't exist in the source. This is /purge for robocopy and –delete for rsync.
create	Passed to checkPath.
silent	Should a progress be printed.

Value

This function is called for its side effect, i.e., a file is copied from to to.

Author(s)

Eliot McIntire and Alex Chubaty

Examples

```
tmpDirFrom <- file.path(tempdir(), "example_fileCopy_from")
tmpDirTo <- file.path(tempdir(), "example_fileCopy_to")
tmpFile1 <- tempfile("file1", tmpDirFrom, ".csv")
tmpFile2 <- tempfile("file2", tmpDirFrom, ".csv")
dir.create(tmpDirFrom, recursive = TRUE, showWarnings = FALSE)
dir.create(tmpDirTo, recursive = TRUE, showWarnings = FALSE)
f1 <- normalizePath(tmpFile1, mustWork = FALSE)
f2 <- normalizePath(tmpFile2, mustWork = FALSE)
t1 <- normalizePath(file.path(tmpDirTo, basename(tmpFile1)), mustWork = FALSE)
t2 <- normalizePath(file.path(tmpDirTo, basename(tmpFile2)), mustWork = FALSE)

write.csv(data.frame(a = 1:10, b = runif(10), c = letters[1:10]), f1)
write.csv(data.frame(c = 11:20, d = runif(10), e = letters[11:20]), f2)
```

```
copyFile(c(f1, f2), c(t1, t2))
file.exists(t1) ## TRUE
file.exists(t2) ## TRUE
identical(read.csv(f1), read.csv(f2)) ## FALSE
identical(read.csv(f1), read.csv(t1)) ## TRUE
identical(read.csv(f2), read.csv(t2)) ## TRUE
```

---

createCache *Low-level functions to create and work with a cache*

Description

These are intended for advanced use only.

Usage

```
createCache(
  cachePath = getOption("reproducible.cachePath"),
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  force = FALSE,
  verbose = getOption("reproducible.verbose")
)

loadFromCache(
  cachePath = getOption("reproducible.cachePath"),
  cacheId,
  preDigest,
  fullCacheTableForObj = NULL,
  cacheSaveFormat = getOption("reproducible.cacheSaveFormat", .rdsFormat),
  .functionName = NULL,
  .dotsFromCache = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose")
)

extractFromCache(sc, elem, ifNot = NULL)

rmFromCache(
  cachePath = getOption("reproducible.cachePath"),
  cacheId,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  cacheSaveFormat = getOption("reproducible.cacheSaveFormat", .rdsFormat),
  verbose = getOption("reproducible.verbose"),
```

```
  ...
)

CacheDBFile(
  cachePath = getOption("reproducible.cachePath"),
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL)
)

CacheStorageDir(cachePath = getOption("reproducible.cachePath"))

CacheStoredFile(
  cachePath = getOption("reproducible.cachePath"),
  cacheId,
  cacheSaveFormat = getOption("reproducible.cacheSaveFormat"),
  obj = NULL,
  readOnly = FALSE
)

CacheDBTableName(
  cachePath = getOption("reproducible.cachePath"),
  drv = getDrv(getOption("reproducible.drv", NULL))
)

CacheIsACache(
  cachePath = getOption("reproducible.cachePath"),
  create = FALSE,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose")
)
```

Arguments

cachePath	A path describing the directory in which to create the database file(s)
drv	A driver, passed to dbConnect
conn	an optional DBIConnection object, as returned by dbConnect().
force	Logical. Should it create a cache in the cachePath, even if it already exists, overwriting.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
cacheId	The cacheId or otherwise digested hash value, as character string.
preDigest	The list of preDigest that comes from CacheDigest of an object
fullCacheTableForObj	The result of showCache, but subsetted for only the cacheId being loaded or selected

cacheSaveFormat	The text string representing the file extension used normally by different save formats; currently only "rds" or "qs" (which now uses qs2 package. Defaults to getOption("reproducible.cacheSaveFormat", "rds")
.functionName	Optional. Used for messaging when this function is called from Cache
.dotsFromCache	Optional. Used internally.
sc	a cache tags data.table object
elem	character string specifying a tagKey value to match
ifNot	character (or NULL) specifying the return value to use if elem not matched
...	Arguments passed to FUN, if FUN is not an expression.
obj	The optional object that is of interest; it may have an attribute "saveRawFile" that would be important.
readOnly	Logical. Only relevant during transition from qs to qs2. Essentially, during transition, qs objects can be read, but not saved. If TRUE then the CacheStoredFile can return a .qs file; if FALSE, then this will not be able to return qs; instead it will return qs2 files.
create	Logical. Currently only affects non **RSQLite** default drivers. If TRUE and there is no Cache database, the function will create one.

Details

- createCache() will create a Cache folder structure and necessary files, based on the particular drv or conn provided;
- loadFromCache() retrieves a single object from the cache, given its cacheId;
- extractFromCache() retrieves a single tagValue from the cache based on the tagKey of elem;
- rmFromCache() removes one or more items from the cache, and updates the cache database files.

Value

- createCache() returns NULL (invisibly) and intended to be called for side effects;
- loadFromCache() returns the object from the cache that has the particular cacheId;
- extractFromCache() returns the tagValue from the cache corresponding to elem if found, otherwise the value of ifNot;
- rmFromCache() returns NULL (invisibly) and is intended to be called for side effects;
- CacheDBFile() returns the name of the database file for a given Cache, when useDBI() == FALSE, or NULL if TRUE;
- CacheDBFiles() (i.e,. plural) returns the name of all the database files for a given Cache when useDBI() == TRUE, or NULL if FALSE;

- `CacheStoredFile()` returns the file path to the file with the specified hash value, This can be loaded to memory with e.g., `loadFile()`.;
- `CacheStorageDir()` returns the name of the directory where cached objects are stored;
- `CacheStoredFile` returns the file path to the file with the specified hash value;
- `CacheDBTableName()` returns the name of the table inside the SQL database, if that is being used;
- `CacheIsACache()` returns a logical indicating whether the `cachePath` is currently a `reproducible` cache database;

Examples

```
data.table::setDTthreads(2)
newCache <- tempdir2()
createCache(newCache)

out <- Cache(rnorm(1), cachePath = newCache)
cacheId <- gsub("cacheId:", "", attr(out, "tags"))
loadFromCache(newCache, cacheId = cacheId)

rmFromCache(newCache, cacheId = cacheId)

# clean up
unlink(newCache, recursive = TRUE)

data.table::setDTthreads(2)
newCache <- tempdir2()

# Given the drv and conn, creates the minimum infrastructure for a cache
createCache(newCache)

CacheDBFile(newCache) # identifies the database file
CacheStorageDir(newCache) # identifies the directory where cached objects are stored

out <- Cache(rnorm(1), cachePath = newCache)
cacheId <- gsub("cacheId:", "", attr(out, "tags"))
CacheStoredFile(newCache, cacheId = cacheId)

# The name of the table inside the SQL database
CacheDBTableName(newCache)

CacheIsACache(newCache) # returns TRUE

# clean up
unlink(newCache, recursive = TRUE)
```

detectActiveCores *Count Active Threads Based on CPU Usage*

Description

This function counts the number of active system processes (threads) that match a given pattern and exceed a specified minimum CPU usage threshold. It works on Unix-like systems (e.g., Linux, macOS) and does not support Windows.

Usage

```
detectActiveCores(pattern = "", minCPU = 50)
```

Arguments

pattern	A character string used to filter process lines. Only processes whose command line matches this pattern will be considered. Default is "" (matches all).
minCPU	A numeric value specifying the minimum CPU usage (in percent) for a process to be considered active. Default is 50.

Value

An integer representing the number of active threads matching the pattern and exceeding the CPU usage threshold. Returns NULL with a message if run on Windows.

Note

This function uses the ps -ef system command and regular expressions to parse CPU usage. It may not be portable across all Unix variants.

Examples

```
detectActiveCores(pattern = "R", minCPU = 30)
```

determineFilename *Determine filename, either automatically or manually*

Description

Determine the filename, given various combinations of inputs.

Usage

```
determineFilename(
  filename2 = NULL,
  filename1 = NULL,
  destinationPath = getOption("reproducible.destinationPath", "."),
  verbose = getOption("reproducible.verbose", 1),
  prefix = "Small",
  ...
)
```

Arguments

filename2	filename2 is optional, and is either NULL (no writing of outputs to disk), or several options for writing the object to disk. If TRUE (the default), it will give it a file name determined by .prefix(basename(filename1), prefix). If a character string, it will use this as its file name. See determineFilename().
filename1	Character strings giving the file paths of the *input* object (filename1) filename1 is only used for messaging (i.e., the object itself is passed in as x) and possibly naming of output (see details and filename2).
destinationPath	Optional. If filename2 is a relative file path, then this will be the directory of the resulting absolute file path.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
prefix	The character string to prepend to filename1, if filename2 not provided.
...	Passed to methods. None currently implemented.

Details

The post processing workflow, which includes this function, addresses several scenarios, and depending on which scenario, there are several file names at play. For example, Raster objects may have file-backed data, and so *possess a file name*, whereas Spatial objects do not. Also, if post processing is part of a prepInputs() workflow, there will always be a file downloaded. From the perspective of postProcess, these are the "inputs" or filename1. Similarly, there may or may not be a desire to write an object to disk after all post processing, filename2.

This subtlety means that there are two file names that may be at play: the "input" file name (filename1), and the "output" filename (filename2). When this is used within postProcess, it is straight forward.

However, when postProcess is used within a prepInputs call, the filename1 file is the file name of the downloaded file (usually automatically known following the downloading, and refered to as targetFile) and the filename2 is the file name of the of post-processed file.

If filename2 is TRUE, i.e., not an actual file name, then the cropped/masked raster will be written to disk with the original filenam1/targetFile name, with prefix prefixed to the basename(targetFile).

If `filename2` is a character string, it will be the path of the saved/written object e.g., passed to `writeOutput`. It will be tested whether it is an absolute or relative path and used as is if absolute or prepended with `destinationPath` if relative.

If `filename2` is `logical`, then the output filename will be `prefix` prefixed to the basename(`filename1`). If a character string, it will be the path returned. It will be tested whether it is an absolute or relative path and used as is if absolute or prepended with `destinationPath` if provided, and if `filename2` is relative.

`downloadRemote` *Download a remote file*

Description

Download a remote file

Usage

```
downloadRemote(
  url,
  archive,
  targetFile,
  checkSums,
  dlFun = NULL,
  fileToDownload,
  messSkipDownload,
  destinationPath,
  overwrite,
  needChecksums,
  .tempPath,
  preDigest,
  alsoExtract = "similar",
  verbose = getOption("reproducible.verbose", 1),
  .callingEnv = parent.frame(),
  ...
)
```

Arguments

`url`	Optional character string indicating the URL to download from. If not specified, then no download will be attempted. If not entry exists in the `CHECKSUMS.txt` (in `destinationPath`), an entry will be created or appended to. This `CHECKSUMS.txt` entry will be used in subsequent calls to `prepInputs` or `preProcess`, comparing the file on hand with the ad hoc `CHECKSUMS.txt`. See table in `preProcess()`.
`archive`	Optional character string giving the path of an archive containing `targetFile`, or a vector giving a set of nested archives (e.g., `c("xxx.tar", "inner.zip", "inner.rar")`). If there is/are (an) inner archive(s), but they are unknown, the

function will try all until it finds the `targetFile`. See table in `preProcess()`. If it is `NA`, then it will *not* attempt to see it as an archive, even if it has archive-like file extension (e.g., `.zip`). This may be useful when an R function is expecting an archive directly.

`targetFile` Character string giving the filename (without relative or absolute path) to the eventual file (raster, shapefile, csv, etc.) after downloading and extracting from a zip or tar archive. This is the file *before* it is passed to `postProcess`. The internal checksumming does not checksum the file after it is `postProcessed` (e.g., cropped/reprojected/masked). Using `Cache` around `prepInputs` will do a sufficient job in these cases. See table in `preProcess()`.

`checkSums` TODO

`dlFun` Optional "download function" name, such as `"raster::getData"`, which does custom downloading, in addition to loading into R. Still experimental.

`fileToDownload` TODO

`messSkipDownload`
The character string text to pass to messaging if download skipped

`destinationPath`
Character string of a directory in which to download and save the file that comes from `url` and is also where the function will look for `archive` or `targetFile`. To prevent repeated downloads in different locations, the user can also set `options("reproducible.dest` to one or more local file paths to search for the file before attempting to download. Default for that option is `NULL` meaning do not search locally. The previous name `options("reproducible.inputPaths")` is still accepted as a backwards-compatible alias.

`overwrite` Logical. Passed to `writeTo` (possibly inside `postProcess`) and `postProcess`.

`needChecksums` Logical indicating whether to generate checksums. ## TODO: add overwrite arg to the function?

`.tempPath` Optional temporary path for internal file intermediate steps. Will be cleared `on.exit` from this function.

`preDigest` The list of `preDigest` that comes from `CacheDigest` of an object

`alsoExtract` Which files to fetch or extract in addition to `targetFile`. Spatial data rarely arrives as one file – a shapefile is five or six, and a raster often has a `.aux.xml`, `.ovr` or `.tfw` alongside carrying its attribute table, overviews or georeferencing – so this controls how much of that company comes along.
The default depends on where the files are coming from, because "everything" only means something when a container bounds it:

`alsoExtract`	**from an** `archive`	**from a** `url`
`NULL` (default)	every file in the archive	as for `"similar"`
`"similar"`	files sharing `targetFile`'s name without its extension	same, discovered by listing the remote directory
`NA` or `"none"`	`targetFile` only	`targetFile` only
a character vector	those files only	those files only

An archive bounds what "all files" can mean, so `NULL` extracts all of it. A `url` has no such bound – the server may hold thousands of unrelated files – so `NULL` there means `targetFile` and its companions, never the whole remote directory.

Each element of a character vector may also be a regular expression: if an element does not match any archive member literally (by relative path or basename), it is passed to `grep()` against the archive's file list and all matching members are extracted. For example, `alsoExtract = "CMD_sm|CMD_sp"` extracts every file whose name contains `CMD_sm` or `CMD_sp`. See table in `preProcess()`.

`verbose` Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., `options('reproducible.verbose' = 0)` to reduce t

`.callingEnv` The environment where the function was called from. Used to find objects, if necessary.

`...` Additional arguments passed to `postProcess()` and `Cache()`. Since `...` is passed to `postProcess()`, these will `...` will also be passed into the inner functions, e.g., `cropInputs()`. Possibly useful other arguments include `dlFun` which is passed to `preProcess`. See details and examples.

`extractFromArchive` *Extract files from archive*

Description

Extract zip or tar archive files, possibly nested in other zip or tar archives.

Usage

```
extractFromArchive(
  archive,
  destinationPath = getOption("reproducible.destinationPath", dirname(archive)),
  neededFiles = NULL,
  extractedArchives = NULL,
  checkSums = NULL,
  needChecksums = 0,
  filesExtracted = character(),
  checkSumFilePath = character(),
  quick = FALSE,
  verbose = getOption("reproducible.verbose", 1),
  .tempPath,
  ...
)
```

Arguments

`archive` Character string giving the path of the archive containing the `file` to be extracted. This path must exist or be `NULL`

`destinationPath`

Character string giving the path where `neededFiles` will be extracted. Defaults to the archive directory.

neededFiles Character string giving the name of the file(s) to be extracted.

extractedArchives
Used internally to track archives that have been extracted from.

checkSums A checksums file, e.g., created by Checksums(..., write = TRUE)

needChecksums A numeric, with 0 indicating do not write a new checksums, 1 write a new one, 2 append new information to existing one.

filesExtracted Used internally to track files that have been extracted.

checkSumFilePath
The full path to the checksum.txt file

quick Passed to Checksums

verbose Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

.tempPath Optional temporary path for internal file intermediate steps. Will be cleared on.exit from this function.

... Passed to unzip or untar, e.g., overwrite

Value

A character vector listing the paths of the extracted archives.

Author(s)

Jean Marchal and Eliot McIntire

Filenames *Return the filename(s) from a* Raster* *object*

Description

This is mostly just a wrapper around filename from the raster package, except that instead of returning an empty string for a RasterStack object, it will return a vector of length >1 for RasterStack.

Usage

```
Filenames(obj, allowMultiple = TRUE, returnList = FALSE)

## S4 method for signature 'ANY'
Filenames(obj, allowMultiple = TRUE, returnList = FALSE)

## S4 method for signature 'environment'
Filenames(obj, allowMultiple = TRUE, returnList = FALSE)
```

```
## S4 method for signature 'list'
Filenames(obj, allowMultiple = TRUE, returnList = FALSE)

## S4 method for signature 'data.table'
Filenames(obj, allowMultiple = TRUE, returnList = FALSE)

## S4 method for signature 'Path'
Filenames(obj, allowMultiple = TRUE, returnList = FALSE)
```

Arguments

obj	A Raster* object (i.e., RasterLayer, RasterStack, RasterBrick)
allowMultiple	Logical. If TRUE, the default, then all relevant filenames will be returned, i.e., in cases such as .grd where multiple files are required. If FALSE, then only the first file will be returned, e.g., filename.grd, in the case of default Raster format in R.
returnList	Default FALSE. If FALSE, then return format will be a character vector. When TRUE, list or environment objects will return a list of character strings or vectors. When returned as a character vector, then the names of objects with >1 filename associated with them will be given a numeric suffix, which means the name in the returned vector does not match the object in the list or environment. When returned as a list, their names are preserved.

Details

New methods can be made for this generic.

Value

A character vector of filenames that are part of the objects passed to obj. This returns NULL is the object is not file-backed or does not have a method to recover the file-backed filename.

Author(s)

Eliot McIntire

---

fixErrorsIn *Fix common errors in GIS layers, using* terra

Description

Currently, this only tests for validity of a SpatVect file, then if there is a problem, it will run terra::makeValid

Usage

```
fixErrorsIn(
  x,
  error = NULL,
  verbose = getOption("reproducible.verbose"),
  fromFnName = ""
)
```

Arguments

x	The SpatStat or SpatVect object to try to fix.
error	The error message, e.g., coming from `try(...)`
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., `options('reproducible.verbose' = 0)` to reduce t
fromFnName	The function name that produced the error, e.g., `maskTo`

Value

An object of the same class as `x`, but with some errors fixed via `terra::makeValid()`

---

gdalProject *3-Step postProcess sequence for SpatRasters using* `gdalwarp`

Description

DEFUNCT: Please use the `postProcessTo` functions.

`gdalResample` is a thin wrapper around `sf::gdal_utils('gdalwarp', ...)` with specific options set, notably, `"-r", "near"`, `-te`, `-te_srs`, `tr`, `-dstnodata = NA`, `-overwrite`.

`gdalMask` is a thin wrapper around `sf::gdal_utils('gdalwarp', ...)` with specific options set, notably, `-cutline`, `-dstnodata = NA`, and `-overwrite`.

Usage

```
gdalProject(
  fromRas,
  toRas,
  filenameDest,
  verbose = getOption("reproducible.verbose"),
  ...
)

gdalResample(
  fromRas,
```

```
  toRas,
  filenameDest,
  verbose = getOption("reproducible.verbose"),
  ...
)

gdalMask(
  fromRas,
  maskToVect,
  writeTo = NULL,
  verbose = getOption("reproducible.verbose"),
  ...
)
```

Arguments

fromRas	see from argument from postProcessTo(), but can only be a SpatRaster.
toRas	see to argument from postProcessTo(), but can only be a SpatRaster.
filenameDest	A filename with an appropriate extension (e.g., .tif) for gdal to write the output to. Since this function is conceived to be part of a chain, and not the final step, this function does not use writeTo, which is reserved for the final step in the chain.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
...	For gdalProject, this can be method. For gdalMask can be destinationPath and touches. For all gdal*, this can also be and datatype.
maskToVect	see maskTo argument from maskTo(), but can only be a SpatVector
writeTo	Optional character string of a filename to use writeRaster to save the final object. Default is NULL, which means there is no writeRaster

Details

gdalProject is a thin wrapper around sf::gdal_utils('gdalwarp', ...) with specific options set, notably, -r to method (in the ...), -t_srs to the crs of the toRas, -te to the extent of the toRas, -te_srs to the crs of the toRas, -dstnodata = NA, and -overwrite.

These three functions were an alternative sequence (gdalProject, gdalResample, gdalMask) for the from + projectTo SpatRaster / maskTo SpatVector case in postProcessTo. The sequence was determined to be faster and more accurate than any other ordering, including running all three steps in one gdalwarp call (one-step gdalwarp resulted in very coarse pixelation when converting from a coarse resolution to fine resolution). They are retained for reference only; the live postProcessTo path no longer dispatches to them.

See Also

gdalResample(), and gdalMask() and the overarching postProcessTo()

Examples

```
if (require("terra", quietly = TRUE)) {
  # prepare dummy data -- 3 SpatRasters, 2 SpatVectors
  # need 2 SpatRaster
  rf <- system.file("ex/elev.tif", package = "terra")
  elev1 <- terra::rast(rf)

  # a polygon vector
  f <- system.file("ex/lux.shp", package = "terra")
  vOrig <- terra::vect(f)
  v <- vOrig[1:2, ]

  # utm <- terra::crs("epsg:23028") # $wkt
  utm <- "+proj=utm +zone=28 +datum=WGS84 +units=m +no_defs"
  vInUTM <- terra::project(vOrig, utm)
  vAsRasInLongLat <- terra::rast(vOrig, resolution = 0.008333333)
  res100 <- 100
  rInUTM <- terra::rast(vInUTM, resolution = res100, vals = 1)
  # crop, reproject, mask, crop a raster with a vector in a different projection
  #  --> gives message about not enough information
  t1 <- postProcessTo(elev1, to = vInUTM)
  # crop, reproject, mask a raster to a different projection, then mask
  t2a <- postProcessTo(elev1, to = vAsRasInLongLat, maskTo = vInUTM)
  t3a <- postProcessTo(elev1, to = rInUTM, maskTo = vInUTM)

}
```

`getRelative` *Relative paths*

Description

Extracting relative file paths.

Usage

```
getRelative(path, relativeToPath)

makeRelative(files, absoluteBase)
```

Arguments

`path`	character vector or list specifying file paths
`relativeToPath`	directory against which `path` will be relativized.
`files`	character vector or list specifying file paths
`absoluteBase`	base directory (as absolute path) to prepend to `files`

Details

- `getRelative()` searches `path` "from the right" (instead of "from the left") and tries to reconstruct it relative to directory specified by `relativeToPath`. This is useful when dealing with symlinked paths.
- `makeRelative()` checks to see if `files` and `normPath(absoluteBase)` share a common path (i.e., "from the left"), otherwise it returns `files`.

Examples

```
## create a project directory (e.g., on a hard drive)
(tmp1 <- tempdir2("myProject", create = TRUE))

## create a cache directory elsewhere (e.g., on an SSD)
(tmp2 <- tempdir2("my_cache", create = TRUE))

## symlink the project cache directory to tmp2
## files created here are actually stored in tmp2
prjCache <- file.path(tmp1, "cache")
file.symlink(tmp2, prjCache)

## create a dummy cache object file in the project cache dir
(tmpf <- tempfile("cache_", prjCache))
cat(rnorm(100), file = tmpf)
file.exists(tmpf)
normPath(tmpf) ## note the 'real' location (i.e., symlink resolved)

getRelative(tmpf, prjCache) ## relative path
getRelative(tmpf, tmp2) ## relative path

makeRelative(tmpf, tmp2) ## abs path; tmpf and normPath(tmp2) don't share common path
makeRelative(tmpf, prjCache) ## abs path; tmpf and normPath(tmp2) don't share common path
makeRelative(normPath(tmpf), prjCache) ## rel path; share common path when both normPath-ed

unlink(tmp1, recursive = TRUE)
unlink(tmp2, recursive = TRUE)
```

`harmonizeCall` *Harmonize all forms of call*

Description

This will convert all known (imagined) calls so that they have the same canonical format i.e., `rnorm(n = 1, mean = 0, sd = 1)`

Usage

```
harmonizeCall(callList, .callingEnv, .functionName = NULL)
```

Arguments

callList	A named list with elements call, usesDots and FUNorig
.callingEnv	The calling environment where Cache was called from
.functionName	A possible function name. If omitted, then it will be deduced from the callList and may be inaccurate.

Value

A named list. We illustrate with the example rnorm(1). The named list will have the original callList (call (the original call, without quote), FUNorig, the original value passed by user to FUN, and usesDots which is a logical indicating whether the ... are used), and appended with new_call (the harmonized call, with the function and arguments evaluated, e.g., (function (n, mean = 0, sd = 1) .Call(C_rnorm, n, mean, sd))(1)), func_call, the same harmonized call with neither function nor arguments not evaluated (e.g., rnorm(1)), func which will be function or method definition function (n, mean = 0, sd = 1) .Call(C_rnorm, n, mean, sd), and .functionName, which will be the function name as a character string (rnorm) either directly passed from the user's .functionName or deduced from the func_call.

---

isUpdated *Has a cached object has been updated?*

Description

Has a cached object has been updated?

Usage

```
isUpdated(x)
```

Arguments

x	cached object

Value

logical

Examples

```
# An object not returned from Cache is not "updated"
isUpdated(1:10)
```

keepOrigGeom *Keep original geometries of* sf *objects*

Description

When intersections occur, what was originally 2 polygons features can become LINESTRING and/or POINT and any COLLECTIONS or MULTI- versions of these. This function evaluates what the original geometry was and drops any newly created *different* geometries. For example, if a POLYGON becomes a COLLECTION of MULTIPOLYGON, POLYGON and POINT geometries, the POINT geometries will be dropped. This function is used internally in postProcessTo().

Usage

```
keepOrigGeom(newObj, origObj)
```

Arguments

newObj	The new, derived sf object
origObj	The previous, object whose geometries should be used.

Value

The original newObj, but with only the type of geometry that entered into the function.

linkOrCopy *Hardlink, symlink, or copy a file*

Description

Attempt first to make a hardlink. If that fails, try to make a symlink (on non-windows systems and symlink = TRUE). If that fails, copy the file.

Usage

```
linkOrCopy(
  from,
  to,
  symlink = TRUE,
  overwrite = TRUE,
  verbose = getOption("reproducible.verbose", 1)
)
```

Arguments

from, to	Character vectors, containing file names or paths. to can alternatively be the path to a single existing directory.
symlink	Logical indicating whether to use symlink (instead of hardlink). Default FALSE.
overwrite	Logical. Passed to writeTo (possibly inside postProcess) and postProcess.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

Value

This function is called for its side effects, which will be a file.link is that is available or file.copy if not (e.g., the two directories are not on the same physical disk).

Note

Use caution with files-backed objects (e.g., rasters). See examples.

Author(s)

Alex Chubaty and Eliot McIntire

See Also

file.link(), file.symlink(), file.copy().

Examples

```
tmpDir <- file.path(tempdir(), "symlink-test")
tmpDir <- normalizePath(tmpDir, winslash = "/", mustWork = FALSE)
dir.create(tmpDir)

f0 <- file.path(tmpDir, "file0.csv")
write.csv(iris, f0)

d1 <- file.path(tmpDir, "dir1")
dir.create(d1)
write.csv(iris, file.path(d1, "file1.csv"))

d2 <- file.path(tmpDir, "dir2")
dir.create(d2)
f2 <- file.path(tmpDir, "file2.csv")

## create link to a file
linkOrCopy(f0, f2)
file.exists(f2) ## TRUE
identical(read.table(f0), read.table(f2)) ## TRUE

## deleting the link shouldn't delete the original file
```

```
unlink(f0)
file.exists(f0) ## FALSE
file.exists(f2) ## TRUE

if (requireNamespace("terra", quietly = TRUE)) {
  ## using spatRasters and other file-backed objects
  f3a <- system.file("ex/test.grd", package = "terra")
  f3b <- system.file("ex/test.gri", package = "terra")
  r3a <- terra::rast(f3a)
  f4a <- file.path(tmpDir, "raster4.grd")
  f4b <- file.path(tmpDir, "raster4.gri")
  linkOrCopy(f3a, f4a) ## hardlink
  linkOrCopy(f3b, f4b) ## hardlink
  r4a <- terra::rast(f4a)

  isTRUE(all.equal(r3a, r4a)) # TRUE

  ## cleanup
  unlink(tmpDir, recursive = TRUE)
}
```

`listGoogleDriveFolder` *List a Google Drive folder, using a mirror listing when remapped*

Description

[Experimental]

A remap-aware alternative to `googledrive::drive_ls()` for *listing* a Google Drive folder's files. When the folder id is in the `reproducible.urlRemap` manifest as a *directory remap* (a `type = "dir"` row; see `makeUrlRemap()`), the files are enumerated from the public bucket listing with **no Google authentication**. Otherwise it falls back to `googledrive::drive_ls()` (which authenticates as usual). It is meant as a near drop-in for the common `as.data.table(drive_ls(url))` pattern: filter the returned `name`s, then pass the corresponding `url`s to `prepInputs()`.

Usage

```
listGoogleDriveFolder(
  url,
  pattern = NULL,
  verbose = getOption("reproducible.verbose", 1)
)
```

Arguments

`url`	A Google Drive folder URL or bare id.
`pattern`	Optional regular expression; only files whose `name` matches are kept (applied to both the mirror and Drive paths).
`verbose`	Numeric verbosity level.

Value

A `data.table` with columns `name` (file name), `id` (the Google Drive file id when listed from Drive; `NA` on the mirror path), and `url` – a URL to hand to `prepInputs()`: the public mirror URL when remapped, otherwise the Drive file URL. A 0-row table when the folder is empty/unreachable.

See Also

`makeUrlRemap()` for the directory-remap manifest; `buckethost::makeMirrorManifest(directories = TRUE)` to build one.

Examples

```
# A directory-remap manifest maps a Drive *folder* id to a bucket prefix-listing
# URL; build it with buckethost::makeMirrorManifest(directories = TRUE), or by
# hand (a `type = "dir"` row):
manifest <- data.frame(
  filename = "2020",
  url = paste0("https://object-arbutus.cloud.computecanada.ca/",
               "predictiveecology/?prefix=SCANFI_v2/2020/&delimiter=/"),
  id = "15T4HIFeqzwp0TuOuxmYoexuXdLFnCZBi",
  type = "dir"
)
remap <- makeUrlRemap(manifest)  # the "dir" row populates the folder-listing map

# Then, with `options(reproducible.urlRemap = manifest)` set, a call such as
#   files <- listGoogleDriveFolder(
#     "https://drive.google.com/drive/folders/15T4HIFeqzwp0TuOuxmYoexuXdLFnCZBi")
# lists that folder from the public mirror with no drive_auth(), returning
# name/id/url columns to feed to prepInputs(); without a manifest it falls back
# to googledrive::drive_ls() (which authenticates as usual).
```

`listNamed` *Create a list with names from object names*

Description

This is a convenience wrapper around `a <- 1; newList <- list(a); names(newList) <- "a"`.

Usage

```
listNamed(...)
```

Arguments

`...`	Any elements to add to a list, as in `base::list`

Details

This will return a named list, where names are the object names, captured internally in the function and assigned to the list. If a user manually supplies names, these will be kept (i.e., not overwritten by the object name).

Examples

```
a <- 1
b <- 2
d <- 3
(newList <- listNamed(a, b, dManual = d)) # "dManual" name kept
```

`makeUrlRemap` *Build a URL remap function from a manifest*

Description

Convenience constructor for the `reproducible.urlRemap` option (see `preProcess()`). Given a `data.frame` with (at least) columns `filename` and `url`, it returns a function `function(url, filename)` suitable for `options(reproducible.urlRemap = ...)`. The returned function matches on the basename of the resolved `filename`: when a manifest row's `filename` matches, its `url` is returned, so the download is redirected there (and, if that URL supports HTTP Range requests, the parallel download path applies). When there is no match it returns `NULL`, so the original URL is kept.

Usage

```
makeUrlRemap(manifest)
```

Arguments

`manifest`	A `data.frame` (or `data.table`) with at least the character columns `filename` and `url`. `filename` is matched against the basename of the file being downloaded. An optional `id` column (Google Drive file id) is matched secondarily, by the id parsed from the Drive `url`, when no filename is available (e.g. an unauthenticated Drive download). An optional `type` column (`"file"`/`"dir"`) marks directory-remap rows (a Drive folder id -> bucket prefix-listing URL), used to enumerate a Drive folder from the public bucket without authentication.

Details

An optional `id` column (also accepted as `googledriveId`, `googledrive_id`, `driveId` or `gid`) holding the Google Drive file id enables a **secondary** match by id: when the resolved `filename` is unavailable — e.g. an unauthenticated session that cannot read a Drive file's metadata — the id is parsed from the Drive `url` and matched against this column. This lets a download be redirected to the (public) mirror with no authentication at all.

An optional type column ("file"/"dir") enables **directory remaps**: a "dir" row maps a Google Drive *folder* id (id column) to a bucket prefix-listing URL (its url). When preProcess() downloads such a folder, it enumerates the folder's files from that public listing instead of googledrive::drive_ls(), so listing the folder needs no authentication. Rows without a type column (or with type = "file") are ordinary file remaps, unchanged. buckethost::makeMirrorManifest(directories = TRUE) emits such a manifest.

The manifest itself — and the responsibility for keeping it current — lives with the user (for example, a community-maintained mirror manifest); reproducible hard-codes no mirror URLs.

Calling makeUrlRemap() yourself is optional: you can also assign the manifest data.frame (or a CSV path/URL) directly to the option, e.g. options(reproducible.urlRemap = read.csv("manifest.csv")), and reproducible will build (and cache) the remap function internally. See the urlRemap entry in reproducibleOptions().

Value

A function of (url, filename) returning a replacement URL, or NULL to keep the original.

See Also

preProcess() for the reproducible.urlRemap option.

Examples

```
manifest <- data.frame(
  filename = "SCANFI_att_biomass_2010_v2_20260119.tif",
  url = paste0(
    "https://object-arbutus.cloud.computecanada.ca/predictiveecology/",
    "SCANFI_v2/2010/SCANFI_att_biomass_2010_v2_20260119.tif"
  )
)
remap <- makeUrlRemap(manifest)

# matched by basename, whatever the original url was
remap("https://drive.google.com/file/d/abc123/view",
      "SCANFI_att_biomass_2010_v2_20260119.tif")
remap("https://example.com/other.tif", "other.tif") # NULL -- keep original

# to apply it to every download in a session:
#   options(reproducible.urlRemap = remap)
```

---

matchCall2 *Remove* quote *and determine if call uses* ...

Description

Minor cleaning up of the FUN and ... to be used subsequently. This does only very minor things as it is run even if useCache = FALSE, i.e., even if the Cache is skipped.

Usage

```
matchCall2(definition, call, envir, envir2 = parent.frame(), FUN)
```

Arguments

definition	a function, by default the function from which match.call is called. See details.
call	an unevaluated call to the function specified by definition, as generated by call.
envir	an environment, from which the ... in call are retrieved, if any.
envir2	Environment. The environment where matchCall2 was called.
FUN	Either a function (e.g., rnorm), a function call (e.g., rnorm(1)), or an unevaluated function call (e.g., using quote()).

Value

A named list with call (the original call, without quote), FUNorig, the original value passed by user to FUN, and usesDots which is a logical indicating whether the ... are used.

---

mergeCache *Merge two cache repositories together*

Description

All the cacheFrom artifacts will be put into cacheTo repository. All userTags will be copied verbatim, including accessed, with 1 exception: date will be the current Sys.time() at the time of merging. The createdDate column will be similarly the current time of merging.

Usage

```
mergeCache(
  cacheTo,
  cacheFrom,
  drvTo = getDrv(getOption("reproducible.drv", NULL)),
  drvFrom = getDrv(getOption("reproducible.drv", NULL)),
  connTo = NULL,
  connFrom = NULL,
  verbose = getOption("reproducible.verbose")
)

## S4 method for signature 'ANY'
mergeCache(
  cacheTo,
  cacheFrom,
  drvTo = getDrv(getOption("reproducible.drv", NULL)),
  drvFrom = getDrv(getOption("reproducible.drv", NULL)),
  connTo = NULL,
```

```
  connFrom = NULL,
  verbose = getOption("reproducible.verbose")
)
```

Arguments

cacheTo	The cache repository (character string of the file path) that will become larger, i.e., merge into this
cacheFrom	The cache repository (character string of the file path) from which all objects will be taken and copied from
drvTo	The database driver for the cacheTo.
drvFrom	The database driver for the cacheFrom
connTo	The connection for the cacheTo. If not provided, then a new one will be made from drvTo and cacheTo
connFrom	The database for the cacheFrom. If not provided, then a new one will be made from drvFrom and cacheFrom
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

Value

The character string of the path of cacheTo, i.e., not the objects themselves.

---

messageDF *Use* message *with a consistent use of* verbose

Description

This family has a consistent use of verbose allowing messages to be turned on or off or verbosity increased or decreased throughout the family of messaging in reproducible.

Usage

```
messageDF(
  df,
  round,
  colour = NULL,
  colnames = NULL,
  indent = NULL,
  verbose = getOption("reproducible.verbose"),
  verboseLevel = 1,
  appendLF = TRUE
)
```

```
messagePrepInputs(
  ...,
  appendLF = TRUE,
  verbose = getOption("reproducible.verbose"),
  verboseLevel = 1
)

messagePreProcess(
  ...,
  appendLF = TRUE,
  verbose = getOption("reproducible.verbose"),
  verboseLevel = 1
)

messageCache(
  ...,
  colour = getOption("reproducible.messageColourCache"),
  verbose = getOption("reproducible.verbose"),
  verboseLevel = 1,
  appendLF = TRUE
)

messageQuestion(..., verboseLevel = 0, appendLF = TRUE)

.messageFunctionFn(
  ...,
  appendLF = TRUE,
  verbose = getOption("reproducible.verbose"),
  verboseLevel = 1
)

messageColoured(
  ...,
  colour = NULL,
  indent = NULL,
  hangingIndent = TRUE,
  verbose = getOption("reproducible.verbose", 1),
  verboseLevel = 1,
  appendLF = TRUE
)
```

Arguments

df	A data.frame, data.table, matrix
round	An optional numeric to pass to round
colour	Any colour that can be understood by cli
colnames	Logical or NULL. If TRUE, then it will print column names even if there aren't any

	in the df (i.e., they will) be V1 etc., NULL will print them if they exist, and FALSE which will omit them.
indent	An integer, indicating whether to indent each line
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
verboseLevel	The numeric value for this message* call, equal or above which verbose must be. The higher this is set, the more unlikely the call will show a message.
appendLF	logical: should messages given as a character string have a newline appended?
...	Any character vector, passed to paste0(...)
hangingIndent	Logical. If there are \n, should there be a handing indent of 2 spaces. Default is TRUE

Details

- messageDF uses message to print a clean square data structure.
- messageColoured allows specific colours to be used.
- messageQuestion sets a high level for verbose so that the message always gets asked.

Value

Used for side effects. This will produce a message of a structured data.frame.

---

minFn *Get min or maximum value of a (Spat)Raster*

Description

During the transition from raster to terra, some functions are not drop in replacements, such as minValue and maxValue became terra::minmax. This helper allows one function to be used, which calls the correct max or min function, depending on whether the object is a Raster or SpatRaster.

Usage

```
minFn(x)

maxFn(x)

dataType2(x, ...)

nlayers2(x)

values2(x, ...)
```

Arguments

x	A Raster or SpatRaster object.
...	Passed to the functions in raster or terra, as needed.

Value

A vector (not matrix as in terra::minmax) with the minimum or maximum value on the Raster or SpatRaster, one value per layer.

Examples

```
if (requireNamespace("terra", quietly = TRUE)) {
  ras <- terra::rast(terra::ext(0, 10, 0, 10), vals = 1:100)
  maxFn(ras)
  minFn(ras)
}
```

---

movedCache *Deal with moved cache issues*

Description

If a user manually copies a complete Cache folder (including the db file and rasters folder), there are issues that must be addressed, depending on the Cache backend used. If using DBI (e.g., RSQLite or Postgres), the db table must be renamed. Run this function after a manual copy of a cache folder. See examples for one way to do that.

Usage

```
movedCache(
  new,
  old,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose")
)
```

Arguments

new	Either the path of the new cachePath where the cache was moved or copied to, or the new DB Table Name
old	Optional, if there is only one table in the new cache path. Either the path of the previous cachePath where the cache was moved or copied from, or the old DB Table Name
drv	If using a database backend, drv must be an object that inherits from DBIDriver (e.g., RSQLite::SQLite).

conn — an optional DBIConnection object, as returned by dbConnect().

verbose — Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

Details

When the backend database for a reproducinle cache is an SQL database, the files on disk cannot be copied manually to a new location because they contain internal tables. Because reproducible gives the main table a name based on the cachePath path, calls to Cache will attempt to call this internally if it detects a name mismatch.

Value

movedCache does not return anything; it is called for its side effects.

Examples

```
data.table::setDTthreads(2)
tmpdir <- "tmpdir"
tmpCache <- "tmpCache"
tmpCacheDir <- normalizePath(file.path(tempdir(), tmpCache), mustWork = FALSE)
tmpdirPath <- normalizePath(file.path(tempdir(), tmpdir), mustWork = FALSE)
bb <- Cache(rnorm, 1, cachePath = tmpCacheDir)

# Copy all files from tmpCache to tmpdir
froms <- normalizePath(dir(tmpCacheDir, recursive = TRUE, full.names = TRUE),
  mustWork = FALSE
)
dir.create(file.path(tmpdirPath, "rasters"), recursive = TRUE, showWarnings = FALSE)
dir.create(file.path(tmpdirPath, "cacheOutputs"), recursive = TRUE, showWarnings = FALSE)
file.copy(
  from = froms, overwrite = TRUE,
  to = gsub(tmpCache, tmpdir, froms)
)

# Can use 'movedCache' to update the database table, though will generally
#   happen automatically, with message indicating so
movedCache(new = tmpdirPath, old = tmpCacheDir)
bb <- Cache(rnorm, 1, cachePath = tmpdirPath) # should recover the previous call
```

---

normPath *Normalize file paths*

Description

Checks the specified path for formatting consistencies:

1. use slash instead of backslash;
2. do tilde etc. expansion;
3. remove trailing slash.

Usage

```
normPath(path)

## S4 method for signature 'character'
normPath(path)

## S4 method for signature 'list'
normPath(path)

## S4 method for signature 'NULL'
normPath(path)

## S4 method for signature 'missing'
normPath()

## S4 method for signature 'logical'
normPath(path)

normPathRel(path)
```

Arguments

path	A character vector of filepaths.

Details

Additionally, `normPath()` attempts to create a absolute paths, whereas `normPathRel()` maintains relative paths.

```
d> getwd()
[1] "/home/achubaty/Documents/GitHub/PredictiveEcology/reproducible"
d> normPathRel("potato/chips")
[1] "potato/chips"
d> normPath("potato/chips")
[1] "/home/achubaty/Documents/GitHub/PredictiveEcology/reproducible/potato/chips"
```

Value

Character vector of cleaned up filepaths.

Examples

```
## normalize file paths
paths <- list("./aaa/zzz",
              "./aaa/zzz/",
              ".//aaa//zzz",
              ".//aaa//zzz/",
              ".\\\\aaa\\\\zzz",
              ".\\\\aaa\\\\zzz\\\\",
              file.path(".", "aaa", "zzz"))

checked <- normPath(paths)
length(unique(checked)) ## 1; all of the above are equivalent

## check to see if a path exists
tmpdir <- file.path(tempdir(), "example_checkPath")

dir.exists(tmpdir) ## FALSE
tryCatch(checkPath(tmpdir, create = FALSE), error = function(e) FALSE) ## FALSE

checkPath(tmpdir, create = TRUE)
dir.exists(tmpdir) ## TRUE

unlink(tmpdir, recursive = TRUE)
```

---

numCoresToUse *Estimate Number of CPU Cores to Use for Parallel Processing*

Description

This function estimates the number of CPU cores that can be safely used for parallel processing, taking into account a minimum threshold, the total number of physical cores, and currently active threads.

Usage

```
numCoresToUse(min = 2, max = NULL)
```

Arguments

`min`	An integer specifying the minimum number of cores to use. Default is `2`.
`max`	An integer specifying the maximum number of cores available, typically the number of physical cores. Default is `max(1L, getOption("Ncpus", 1L), parallel::detectCores() - 1, logical = FALSE, na.rm = TRUE)`.

Value

An integer representing the number of cores that can be used for parallel tasks, ensuring at least `min` cores are used, while subtracting one for the current process and an estimate of actively used threads (via `detectActiveCores()`).

Note

This function depends on `detectActiveCores()` and is not supported on Windows systems.

See Also

`detectActiveCores()`

Examples

```
if (FALSE) {
  numCoresToUse()
  numCoresToUse(min = 4)
}
```

---

objSize *Wrapper around* `lobstr::obj_size`

Description

This function attempts to estimate the real object size of an object. If the object has pass-by-reference semantics, it may not estimate the object size well without a specific method developed. For the case of `terra` class objects, this will be accurate (both RAM and file size), but only if it is not passed inside a list or environment. To get an accurate size of these, they should be passed individually.

Usage

```
objSize(x, quick = FALSE, recursive = FALSE, ...)

objSizeSession(sumLevel = Inf, enclosingEnvs = TRUE, .prevEnvirs = list())
```

Arguments

`x`	An object
`quick`	Logical. If `FALSE`, then an attribute, "objSize" will be added to the returned value, with each of the elements' object size returned also.
`recursive`	Logical. If `TRUE`, then, in addition to evaluating the whole object, it will also return the recursive sizes of the elements of a list or environment.
`...`	Additional arguments (currently unused), enables backwards compatible use.
`sumLevel`	Numeric, indicating at which depth in the list of objects should the object sizes be summed (summarized). Default is `Inf`, meaning no sums. Currently, the only option other than Inf is 1: `objSizeSession(1)`, which gives the size of each package.
`enclosingEnvs`	Logical indicating whether to include enclosing environments. Default `TRUE`.
`.prevEnvirs`	For internal account keeping to identify and prevent duplicate counting

Details

For functions, a user can include the enclosing environment as described https://www.r-bloggers.com/2015/03/using-closures-as-objects-in-r/ and http://adv-r.had.co.nz/memory.html. It is not entirely clear which estimate is better. However, if the enclosing environment is the .GlobalEnv, it will not be included even though enclosingEnvs = TRUE.

objSizeSession will give the size of the whole session, including loaded packages. Because of the difficulties in calculating the object size of base and methods packages and Autoloads, these are omitted.

Value

This will return the result from lobstr::obj_size, i.e., a lobstr_bytes which is a numeric. If quick = FALSE, it will also have an attribute, "objSize", which will be a list with each element being the objSize of the individual elements of x. This is particularly useful if x is a list or environment. However, because of the potential for shared memory, the sum of the individual elements will generally not equal the value returned from this function.

Examples

```
library(utils)

foo <- new.env()
foo$b <- 1:10
foo$d <- 1:10

objSize(foo) # all the elements in the environment
utils::object.size(foo) # different - only measuring the environment as an object

utils::object.size(prepInputs) # only the function, without its enclosing environment
objSize(prepInputs) # the function, plus its enclosing environment

os1 <- utils::object.size(as.environment("package:reproducible"))
(os1) # very small -- just the environment container
```

---

paddedFloatToChar *Convert numeric to character with padding*

Description

This will pad floating point numbers, right or left. For integers, either class integer or functionally integer (e.g., 1.0), it will not pad right of the decimal. For more specific control or to get exact padding right and left of decimal, try the stringi package. It will also not do any rounding. See examples.

Usage

```
paddedFloatToChar(x, padL = ceiling(log10(x + 1)), padR = 3, pad = "0")
```

Arguments

x	numeric. Number to be converted to character with padding
padL	numeric. Desired number of digits on left side of decimal. If not enough, pad will be used to pad.
padR	numeric. Desired number of digits on right side of decimal. If not enough, pad will be used to pad.
pad	character to use as padding (nchar(pad) == 1 must be TRUE).

Value

Character string representing the filename.

Author(s)

Eliot McIntire and Alex Chubaty

Examples

```
paddedFloatToChar(1.25)
paddedFloatToChar(1.25, padL = 3, padR = 5)
paddedFloatToChar(1.25, padL = 3, padR = 1) # no rounding, so keeps 2 right of decimal
```

Path-class *Coerce a character string to a class "Path"*

Description

Allows a user to specify that their character string is indeed a filepath. Thus, methods that require only a filepath can be dispatched correctly.

Usage

```
asPath(obj, nParentDirs = 0)

## S3 method for class 'character'
asPath(obj, nParentDirs = 0)

## S3 method for class '`NULL`'
asPath(obj, nParentDirs = 0)

## S4 method for signature 'character,Path'
coerce(from, to = "Path", strict = TRUE)
```

Arguments

obj	A character string to convert to a Path.
nParentDirs	A numeric indicating the number of parent directories starting from basename(obj) = 0 to keep for the digest
from	A character string to coerce to a Path object.
to	The target class, "Path".
strict	Logical; passed through by the S4 coercion machinery.

Details

It is often difficult or impossible to know algorithmically whether a character string corresponds to a valid filepath. In the case where it is en existing file, file.exists can work. But if it does not yet exist, e.g., for a save, it is difficult to know whether it is a valid path before attempting to save to the path.

This function can be used to remove any ambiguity about whether a character string is a path. It is primarily useful for achieving repeatability with Caching. Essentially, when Caching, arguments that are character strings should generally be digested verbatim, i.e., it must be an exact copy for the Cache mechanism to detect a candidate for recovery from the cache. Paths, are different. While they are character strings, there are many ways to write the same path. Examples of identical meaning, but different character strings are: path expanding of ~ vs. not, double back slash vs. single forward slash, relative path vs. absolute path. All of these should be assessed for their actual file or directory location, NOT their character string. By converting all character string that are actual file or directory paths with this function, then Cache will correctly assess the location, NOT the character string representation.

Value

A vector of class Path, which is similar to a character, but has an attribute indicating how deep the Path should be considered "digestible". In other words, most of the time, only some component of an absolute path is relevant for evaluating its purpose in a Cache situation. In general, this is usually equivalent to just the "relative" path

Examples

```
tmpf <- tempfile(fileext = ".csv")
file.exists(tmpf) ## FALSE
tmpfPath <- asPath(tmpf)
is(tmpf, "Path") ## FALSE
is(tmpfPath, "Path") ## TRUE
```

postProcess *Generic function to post process objects*

Description

[Stable]

The method for GIS objects (terra `Spat*` & sf classes) will crop, reproject, and mask, in that order. This is a wrapper for `cropTo()`, `fixErrorsIn()`, `projectTo()`, `maskTo()` and `writeTo()`, with a required amount of data manipulation between these calls so that the crs match.

Usage

```
postProcess(x, ...)

## S3 method for class 'list'
postProcess(x, ...)

## Default S3 method:
postProcess(x, ...)
```

Arguments

`x`	A GIS object of postProcessing, e.g., Spat* or sf*. This can be provided as a `rlang::quosure` or a normal R object.
`...`	Additional arguments passed to methods. For `spatialClasses`, these are: `cropTo()`, `fixErrorsIn()`, `projectTo()`, `maskTo()`, `determineFilename()`, and `writeTo()`. Each of these may also pass `...` into other functions, like `writeTo()`. This might include potentially important arguments like `datatype`, `format`. Also passed to `terra::project`, with likely important arguments such as `method = "bilinear"`. See details.

Value

A GIS file (e.g., `RasterLayer`, `SpatRaster` etc.) that has been appropriately cropped, reprojected, masked, depending on the inputs.

Post processing sequence

If the `rasterToMatch` or `studyArea` are passed, then the following sequence will occur:

1. Fix errors `fixErrorsIn()`. Currently only errors fixed are for `SpatialPolygons` using `buffer(..., width = 0)`.
2. Crop using `cropTo()`
3. Project using `projectTo()`
4. Mask using `maskTo()`
5. Determine file name `determineFilename()`

6. Write that file name to disk, optionally `writeTo()`

NOTE: checksumming does not occur during the post-processing stage, as there are no file downloads. To achieve fast results, wrap `prepInputs` with `Cache`

Backwards compatibility with `rasterToMatch` and/or `studyArea` arguments

For backwards compatibility, `postProcess` will continue to allow passing `rasterToMatch` and/or `studyArea` arguments. Depending on which of these are passed, different things will happen to the `targetFile` located at `filename1`.

See *Use cases* section in `postProcessTo()` for post processing behaviour with the new `from` and `to` arguments.

If `targetFile` is a raster (`Raster*`, or `SpatRaster`) object::

	rasterToMatch	studyArea	Both
extent	Yes	Yes	rasterToMatch
resolution	Yes	No	rasterToMatch
projection	Yes	No*	rasterToMatch*
alignment	Yes	No	rasterToMatch
mask	No**	Yes	studyArea**

*Can be overridden with `useSAcrs`.

**Will mask with `NA`s from `rasterToMatch` if `maskWithRTM`.

If `targetFile` is a vector (`Spatial*`, `sf` or `SpatVector`) object::

	rasterToMatch	studyArea	Both
extent	Yes	Yes	rasterToMatch
resolution	NA	NA	NA
projection	Yes	No*	rasterToMatch*
alignment	NA	NA	NA
mask	No	Yes	studyArea

*Can be overridden with `useSAcrs`

See Also

`prepInputs`

Examples

```
if (requireNamespace("terra", quietly = TRUE) &&
    requireNamespace("withr", quietly = TRUE)) {
  library(reproducible)
  withr::local_dir(withr::local_tempdir())
  withr::local_options(reproducible.inputPaths = NULL)
  # od <- setwd(tempdir2())
```

```
# download a (spatial) file from remote url (which often is an archive) load into R
# need 3 files for this example; 1 from remote, 2 local
dPath <- file.path(tempdir2())
# the `CRAN` tag tracks terra's current CRAN release; `master` moves and is flakier
remoteTifUrl <- "https://github.com/rspatial/terra/raw/CRAN/inst/ex/elev.tif"

localFileLuxSm <- system.file("ex/luxSmall/luxSmall.shp", package = "reproducible")
localFileLux <- system.file("ex/lux.shp", package = "terra")

# 1 step for each layer
# 1st step -- get study area
studyArea <- prepInputs(localFileLuxSm, fun = "terra::vect") # default is sf::st_read

# Requires internet -- gated on NOT_CRAN so CRAN's machines are never asked to
#   reach a third-party server, and wrapped in try() for when the server is down
if (identical(Sys.getenv("NOT_CRAN"), "true")) {
  # 2nd step: make the input data layer like the studyArea map
  elevForStudy <- try(prepInputs(url = remoteTifUrl, to = studyArea, res = 250,
                                 destinationPath = dPath, useCache = FALSE))
}

## The single call above is equivalent to all the steps below, which must be known and
##   repeated for every layer. Left commented out: it is an explanation, not a test.
##
## dir.create(dPath, recursive = TRUE, showWarnings = FALSE)
## file.copy(localFileLuxSm, file.path(dPath, basename(localFileLuxSm)))
## studyArea2 <- terra::vect(localFileLuxSm)
## if (!all(terra::is.valid(studyArea2))) studyArea2 <- terra::makeValid(studyArea2)
## tf <- tempfile(fileext = ".tif")
## download.file(url = remoteTifUrl, destfile = tf, mode = "wb", quiet = TRUE)
## Checksums(dPath, write = TRUE, files = tf)
## elevOrig <- terra::rast(tf)
## studyAreaCrs <- terra::crs(studyArea)
## # Build an explicit target raster (same CRS as studyArea, res = 250) and project to
## # it. Avoids the recursive `terra::project(x, char_crs, res = N)` shorthand that
## # recent terra (~1.9) regressed with `[write] unknown option(s): xscale,yscale`.
## elevTarget <- terra::project(terra::rast(elevOrig), studyAreaCrs)
## elevTarget <- terra::rast(terra::ext(elevTarget),
##                           crs = terra::crs(elevTarget),
##                           resolution = 250)
## elevForStudy2 <- terra::project(elevOrig, elevTarget) |>
##   terra::mask(studyArea2) |>
##   terra::crop(studyArea2)
##
## isTRUE(all.equal(elevForStudy, elevForStudy2)) # TRUE!

# sf class
if (requireNamespace("sf", quietly = TRUE)) {
  studyAreaSmall <- prepInputs(localFileLuxSm, fun = "sf::st_read")
  studyAreas <- list()
  studyAreas[["orig"]] <- prepInputs(localFileLux)
  studyAreas[["reprojected"]] <- projectTo(studyAreas[["orig"]], studyAreaSmall)
 studyAreas[["cropped"]] <- suppressWarnings(cropTo(studyAreas[["orig"]], studyAreaSmall))
```

```
 studyAreas[["masked"]] <- suppressWarnings(maskTo(studyAreas[["orig"]], studyAreaSmall))
}

# SpatVector-- note: doesn't matter what class the "to" object is, only the "from"
studyAreas <- list()
studyAreaSmall <- prepInputs(localFileLuxSm)
studyAreas[["orig"]] <- prepInputs(localFileLux)
studyAreas[["reprojected"]] <- projectTo(studyAreas[["orig"]], studyAreaSmall)
studyAreas[["cropped"]] <- suppressWarnings(cropTo(studyAreas[["orig"]], studyAreaSmall))
studyAreas[["masked"]] <- suppressWarnings(maskTo(studyAreas[["orig"]], studyAreaSmall))
if (interactive()) {
  par(mfrow = c(2,2));
  out <- lapply(studyAreas, function(x) terra::plot(x))
}

withr::deferred_run()
# setwd(od)
}
```

---

postProcessInputsSuperseded

Superseded postProcess *input functions*

Description

[Superseded]

Development on `cropInputs()`, `projectInputs()`, and `maskInputs()` is complete. For new code, use `cropTo()`, `projectTo()`, and `maskTo()` respectively (or the all-in-one `postProcessTo()`). These functions are retained and continue to work by delegating to their successors.

Usage

```
cropInputs(
  x,
  studyArea,
  rasterToMatch,
  verbose = getOption("reproducible.verbose", 1),
  ...
)

## Default S3 method:
cropInputs(x, ...)

projectInputs(
  x,
  targetCRS,
  verbose = getOption("reproducible.verbose", 1),
  ...
```

```
)

## Default S3 method:
projectInputs(x, targetCRS, ...)

maskInputs(x, studyArea, ...)

## Default S3 method:
maskInputs(
  x,
  studyArea,
  rasterToMatch = NULL,
  maskWithRTM = NULL,
  verbose = getOption("reproducible.verbose", 1),
  ...
)
```

Arguments

x	An object to do a geographic raster::mask/raster::intersect. See methods.
studyArea	SpatialPolygons* object used for masking and possibly cropping if no rasterToMatch is provided. If not in same CRS, then it will be spTransformed to CRS of x before masking. Currently, this function will not reproject the x. Optional in postProcess.
rasterToMatch	Template Raster* object passed to the to argument of projectTo(), thus will changing the resolution and projection of x. See details in postProcessTo().
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
...	Passed to methods. None currently implemented.
targetCRS	The CRS of x at the end of this function (i.e., the goal)
maskWithRTM	Logical. If TRUE, the rasterToMatch will be used as the masking object (masking with its NA values); otherwise studyArea is used.

Details

cropInputs() crops a Spatial* or Raster* object. If used, all arguments will be passed to cropTo anyway.

projectInputs() reprojects a Raster*, Spatial*, or sf object.

maskInputs() masks a raster or vector object to a study area.

Value

A GIS file (e.g., RasterLayer, SpatRaster etc.) that has been appropriately cropped.

A file of the same type as starting, but with projection (and possibly other characteristics, including resolution, origin, extent if changed).

A GIS file (e.g., RasterLayer, SpatRaster etc.) that has been appropriately reprojected.

A GIS file (e.g., RasterLayer, SpatRaster etc.) that has been appropriately masked.

Author(s)

Eliot McIntire, Jean Marchal, Ian Eddy, and Tati Micheletti

Eliot McIntire and Jean Marchal

See Also

`projectTo()`

`maskTo()`, `postProcessTo()` for related examples

`postProcessTo` *Transform a GIS dataset so it has the properties (extent, projection, mask) of another*

Description

This function provides a single step to achieve the GIS operations "pre-crop-with-buffer-to-speed-up-projection", "project", "post-projection-crop", "mask" and possibly "write". It uses primarily the `terra` package internally (with some minor functions from `sf`) in an attempt to be as efficient as possible, except if all inputs are `sf` objects. (in which case `sf` is used). Currently, this function is tested with `sf`, `SpatVector`, `SpatRaster`, `Raster*` and `Spatial*` objects passed to `from`, and the same plus `SpatExtent`, and `crs` passed to `to` or the relevant `*to` functions. For this function, Gridded means a `Raster*` class object from `raster` or a `SpatRaster` class object from `terra`. Vector means a `Spatial*` class object from `sp`, a `sf` class object from `sf`, or a `SpatVector` class object from `terra`. This function is also used internally with the deprecated family `postProcess()`, `*Inputs`, such as `cropInputs()`.

Usage

```
postProcessTo(
  from,
  to,
  cropTo = NULL,
  projectTo = NULL,
  maskTo = NULL,
  writeTo = NULL,
  overwrite = TRUE,
  verbose = getOption("reproducible.verbose"),
  ...
)

postProcessTerra(
  from,
```

```
  to,
  cropTo = NULL,
  projectTo = NULL,
  maskTo = NULL,
  writeTo = NULL,
  overwrite = TRUE,
  verbose = getOption("reproducible.verbose"),
  ...
)

maskTo(
  from,
  maskTo,
  overwrite = FALSE,
  verbose = getOption("reproducible.verbose"),
  ...
)

projectTo(
  from,
  projectTo,
  overwrite = FALSE,
  verbose = getOption("reproducible.verbose"),
  ...
)

cropTo(
  from,
  cropTo = NULL,
  needBuffer = FALSE,
  overwrite = FALSE,
  verbose = getOption("reproducible.verbose"),
  ...
)

writeTo(
  from,
  writeTo,
  overwrite = getOption("reproducible.overwrite"),
  isStack = NULL,
  isBrick = NULL,
  isRaster = NULL,
  isSpatRaster = NULL,
  verbose = getOption("reproducible.verbose"),
  ...
)
```

Arguments

from	A Gridded or Vector dataset on which to do one or more of: crop, project, mask, and write
to	A Gridded or Vector dataset which is the object whose metadata will be the target for cropping, projecting, and masking of from.
cropTo	Optional Gridded or Vector dataset which, if supplied, will supply the extent with which to crop from. To omit cropping completely, set this to NA. If supplied, this will override to for the cropping step. Defaults to NULL, which means use to
projectTo	Optional Gridded or Vector dataset, or crs object (e.g., sf::st_crs). If Gridded it will supply the crs, extent, res, and origin to project the from to. If Vector, it will provide the crs only. The resolution and extent will be taken from res(from) (i.e. ncol(from)*nrow(from)). If a Vector, the extent of the projectTo is not used (unless it is also passed to cropTo. To omit projecting, set this to NA. If supplied, this will override to for the projecting step. Defaults to NULL, which means use to. **Attention.** Conflicts may arise with when projectTo is a Vector/CRS object with a distinct CRS from to. Because to is used for masking *after* from is re-projected using projectTo, the extents of to and from may no longer overlap (as in *align*) perfectly leading to failure during the masking step. We recommend passing a raster templates to projectTo whose extent and CRS are both compatible with the object used later for masking (either to or maskTo).
maskTo	Optional Gridded or Vector dataset which, if supplied, will supply the extent with which to mask from. If Gridded, it will mask with the NA values on the maskTo; if Vector, it will mask on the terra::aggregate(maskTo). To omit masking completely, set this to NA. If supplied, this will override to for the masking step. Defaults to NULL, which means use to
writeTo	Optional character string of a filename to use writeRaster to save the final object. Default is NULL, which means there is no writeRaster
overwrite	Logical. Used if writeTo is not NULL; also if terra determines that the object requires writing to disk during a crop, mask or project call e.g., because it is too large.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
...	Arguments passed to terra::mask (for maskTo), terra::project (for projectTo) or terra::writeRaster (for writeTo) and not used for cropTo, as well postProcess's rasterToMatch and studyArea arguments (see below). Commonly used arguments might be method, touches, and datatype. If filename is passed, it will be ignored; use writeTo = .
needBuffer	Logical. Defaults to FALSE, meaning nothing is done out of the ordinary. If TRUE, then a buffer around the cropTo, so that if a reprojection has to happen on the cropTo prior to using it as a crop layer, then a buffer of 1.5 * res(cropTo) will occur prior, so that no edges are cut off.

isStack, isBrick, isRaster, isSpatRaster
: Logical. Default NULL. Used to convert from back to these classes prior to writing, if provided.

Details

postProcessTo is a wrapper around (an initial "wide" crop for speed) cropTo(needBuffer = TRUE), projectTo, cropTo (the actual crop for precision), maskTo, writeTo. Users can call each of these individually.

postProcessTerra is the early name of this function that is now postProcessTo.

This function is meant to replace postProcess() with the more efficient and faster terra functions.

Value

An object of the same class as from, but potentially cropped (via cropTo()), projected (via projectTo()), masked (via maskTo()), and written to disk (via writeTo()).

Use Cases

The table below shows what will result from passing different classes to from and to:

from	to	from will have:
Gridded	Gridded	the extent, projection, origin, resolution and masking where there are NA from the to
Gridded	Vector	the projection, origin, and mask from to, and extent will be a round number of pixels that fit within the
Vector	Vector	the projection, origin, extent and mask from to

If one or more of the *To arguments are supplied, these will override individual components of to. If to is omitted or NULL, then only the *To arguments that are used will be performed. In all cases, setting a *To argument to NA will prevent that step from happening.

projectTo

Since these functions use the gis capabilities of sf and terra, they will only be able to do things that those functions can do. One key caution, which is stated clearly in ?terra::project is that projection of a raster (i.e., gridded) object should always be with another gridded object. If the user chooses to supply a projectTo that is a vector object for a from that is gridded, there may be unexpected failures due e.g., to extents not overlapping during the maskTo stage.

Backwards compatibility with postProcess

rasterToMatch **and** studyArea**::**

If these are supplied, postProcessTo will use them instead of to. If only rasterToMatch is supplied, it will be assigned to to. If only studyArea is supplied, it will be used for cropTo and maskTo; it will only be used for projectTo if useSAcrs = TRUE. If both rasterToMatch and studyArea are supplied, studyArea will only be applied to maskTo (unless maskWithRTM = TRUE), and, optionally, to projectTo (if useSAcrs = TRUE); everything else will be from rasterToMatch.

targetCRS, filename2, useSAcrs, maskWithRTM::
targetCRS if supplied will be assigned to projectTo. filename2 will be assigned to writeTo. If useSAcrs is set, then the studyArea will be assigned to projectTo. If maskWithRTM is used, then the rasterToMath will be assigned to maskTo. All of these will override any existing values for these arguments.

See also postProcess() documentation section on *Backwards compatibility with* rasterToMatch *and/or* studyArea for further detail.

Cropping

If cropTo is not NA, postProcessTo does cropping twice, both the first and last steps. It does it first for speed, as cropping is a very fast algorithm. This will quickly remove a bunch of pixels that are not necessary. But, to not create bias, this first crop is padded by `2 * res(from)[1]`), so that edge cells still have a complete set of neighbours. The second crop is at the end, after projecting and masking. After the projection step, the crop is no longer tight. Under some conditions, masking will effectively mask and crop in one step, but under some conditions, this is not true, and the mask leaves padded NAs out to the extent of the from (as it is after crop, project, mask). Thus the second crop removes all NA cells so they are tight to the mask.

See Also

maskTo(), cropTo(), projectTo(), writeTo(), and fixErrorsIn(). Also the functions that call sf::gdal_utils(...) directly: gdalProject(), gdalResample(), gdalMask()

Examples

```
if (require("terra", quietly = TRUE)) {
  # prepare dummy data -- 3 SpatRasters, 2 SpatVectors
  # need 2 SpatRaster
  rf <- system.file("ex/elev.tif", package = "terra")
  elev1 <- terra::rast(rf)

  # a polygon vector
  f <- system.file("ex/lux.shp", package = "terra")
  vOrig <- terra::vect(f)
  v <- vOrig[1:2, ]

  # utm <- terra::crs("epsg:23028") # $wkt
  utm <- "+proj=utm +zone=28 +datum=WGS84 +units=m +no_defs"
  vInUTM <- terra::project(vOrig, utm)
  vAsRasInLongLat <- terra::rast(vOrig, resolution = 0.008333333)
  res100 <- 100
  rInUTM <- terra::rast(vInUTM, resolution = res100, vals = 1)
  # crop, reproject, mask, crop a raster with a vector in a different projection
  #  --> gives message about not enough information
  t1 <- postProcessTo(elev1, to = vInUTM)
  # crop, reproject, mask a raster to a different projection, then mask
  t2a <- postProcessTo(elev1, to = vAsRasInLongLat, maskTo = vInUTM)
  t3a <- postProcessTo(elev1, to = rInUTM, maskTo = vInUTM)

}
```

prepInputs *Download and optionally post-process files*

Description

[Stable]

This function can be used to prepare R objects from remote or local data sources. The object of this function is to provide a reproducible version of a series of commonly used steps for getting, loading, and processing data. This function has two stages: Getting data (download, extracting from archives, loading into R) and post-processing (for `Spatial*` and `Raster*` objects, this is crop, reproject, mask/intersect). To trigger the first stage, provide `url` or `archive`. To trigger the second stage, provide `studyArea` or `rasterToMatch`. See examples.

Usage

```
prepInputs(
  targetFile = NULL,
  url = NULL,
  archive = NULL,
  alsoExtract = NULL,
  destinationPath = getOption("reproducible.destinationPath", "."),
  fun = NULL,
  quick = getOption("reproducible.quick"),
  overwrite = getOption("reproducible.overwrite", FALSE),
  purge = FALSE,
  useCache = getOption("reproducible.useCache", 2),
  .tempPath,
  verbose = getOption("reproducible.verbose", 1),
  ...
)
```

Arguments

`targetFile`	Character string giving the filename (without relative or absolute path) to the eventual file (raster, shapefile, csv, etc.) after downloading and extracting from a zip or tar archive. This is the file *before* it is passed to `postProcess`. The internal checksumming does not checksum the file after it is `postProcessed` (e.g., cropped/reprojected/masked). Using `Cache` around `prepInputs` will do a sufficient job in these cases. See table in `preProcess()`.
`url`	Optional character string indicating the URL to download from. If not specified, then no download will be attempted. If not entry exists in the `CHECKSUMS.txt` (in `destinationPath`), an entry will be created or appended to. This `CHECKSUMS.txt` entry will be used in subsequent calls to `prepInputs` or `preProcess`, comparing the file on hand with the ad hoc `CHECKSUMS.txt`. See table in `preProcess()`.
`archive`	Optional character string giving the path of an archive containing `targetFile`, or a vector giving a set of nested archives (e.g., `c("xxx.tar", "inner.zip",`

"inner.rar")). If there is/are (an) inner archive(s), but they are unknown, the function will try all until it finds the targetFile. See table in preProcess(). If it is NA, then it will *not* attempt to see it as an archive, even if it has archive-like file extension (e.g., .zip). This may be useful when an R function is expecting an archive directly.

alsoExtract — Which files to fetch or extract in addition to targetFile. Spatial data rarely arrives as one file – a shapefile is five or six, and a raster often has a .aux.xml, .ovr or .tfw alongside carrying its attribute table, overviews or georeferencing – so this controls how much of that company comes along.

The default depends on where the files are coming from, because "everything" only means something when a container bounds it:

alsoExtract	**from an** archive	**from a** url
NULL (default)	every file in the archive	as for "similar"
"similar"	files sharing targetFile's name without its extension	same, discovered by listing the remote directory
NA or "none"	targetFile only	targetFile only
a character vector	those files only	those files only

An archive bounds what "all files" can mean, so NULL extracts all of it. A url has no such bound – the server may hold thousands of unrelated files – so NULL there means targetFile and its companions, never the whole remote directory.

Each element of a character vector may also be a regular expression: if an element does not match any archive member literally (by relative path or basename), it is passed to grep() against the archive's file list and all matching members are extracted. For example, alsoExtract = "CMD_sm|CMD_sp" extracts every file whose name contains CMD_sm or CMD_sp. See table in preProcess().

destinationPath

Character string of a directory in which to download and save the file that comes from url and is also where the function will look for archive or targetFile. To prevent repeated downloads in different locations, the user can also set options("reproducible.dest to one or more local file paths to search for the file before attempting to download. Default for that option is NULL meaning do not search locally. The previous name options("reproducible.inputPaths") is still accepted as a backwards-compatible alias.

fun — Optional. If specified, this will attempt to load whatever file was downloaded during preProcess via dlFun. This can be either a function (e.g., sf::st_read), character string (e.g., "base::load"), NA (for no loading, useful if dlFun already loaded the file) or if extra arguments are required in the function call, it must be a call naming targetFile (e.g., sf::st_read(targetFile, quiet = TRUE)) as the file path to the file to load. See details and examples below.

quick — Logical. This is passed internally to Checksums() (the quickCheck argument), and to Cache() (the quick argument). This results in faster, though less robust checking of inputs. See the respective functions.

overwrite — Logical. Passed to writeTo (possibly inside postProcess) and postProcess.

purge — Logical or Integer. 0/FALSE (default) keeps existing CHECKSUMS.txt file and prepInputs will write or append to it. 1/TRUE will deleted the entire CHECKSUMS.txt file. Other options, see details.

useCache	Passed to Cache in various places. Defaults to getOption("reproducible.useCache", 2L) in prepInputs, and getOption("reproducible.useCache", FALSE) if calling any of the inner functions manually. For prepInputs, this mean it will use Cache only up to 2 nested levels, which includes preProcess. postProcess and its nested *Input functions (e.g., cropInputs, projectInputs, maskInputs) are no longer internally cached, as terra processing speeds mean internal caching is more time consuming. We recommend caching the full prepInputs call instead (e.g. prepInputs(...) |> Cache()).
.tempPath	Optional temporary path for internal file intermediate steps. Will be cleared on.exit from this function.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
...	Additional arguments passed to postProcess() and Cache(). Since ... is passed to postProcess(), these will ... will also be passed into the inner functions, e.g., cropInputs(). Possibly useful other arguments include dlFun which is passed to preProcess. See details and examples.

Value

This is an omnibus function that will return an R object that will have resulted from the running of preProcess() and postProcess() or postProcessTo(). Thus, if it is a GIS object, it may have been cropped, reprojected, "fixed", masked, and written to disk.

Stage 1 - Getting data

See preProcess() for combinations of arguments.

1. Download from the web via either googledrive::drive_download(), utils::download.file();
2. Extract from archive using unzip() or untar();
3. Load into R using terra::rast, sf::st_read, or any other function passed in with fun;
4. Checksumming of all files during this process. This is put into a 'CHECKSUMS.txt' file in the destinationPath, appending if it is already there, overwriting the entries for same files if entries already exist.

Stage 2 - Post processing

This will be triggered if either rasterToMatch or studyArea is supplied.

1. Fix errors. Currently only errors fixed are for SpatialPolygons using buffer(..., width = 0);
2. Crop using cropTo();
3. Project using projectTo();
4. Mask using maskTo();
5. write the file to disk via writeTo().

NOTE: checksumming does not occur during the post-processing stage, as there are no file downloads. To achieve fast results, wrap `prepInputs` with `Cache`.

postProcessing of `Spat*`, `sf`, `Raster*` and `Spatial*` objects::

The following has been DEPRECATED because there are a sufficient number of ambiguities that this has been changed in favour of `from` and the `*to` family. See `postProcessTo()`.

DEPRECATED: If `rasterToMatch` or `studyArea` are used, then this will trigger several subsequent functions, specifically the sequence, *Crop, reproject, mask*, which appears to be a common sequence while preparing spatial data from diverse sources. See `postProcess()` documentation section on *Backwards compatibility with* `rasterToMatch` *and/or* `studyArea` *arguments* to understand various combinations of `rasterToMatch` and/or `studyArea`.

`fun`

`fun` offers the ability to pass any custom function with which to load the file obtained by `preProcess` into the session. There are two cases that are dealt with: when the `preProcess` downloads a file (including via `dlFun`), `fun` must deal with a file; and, when `preProcess` creates an R object (e.g., raster::getData returns an object), `fun` must deal with an object.

`fun` can be supplied in three ways: a function, a character string (i.e., a function name as a string), or an expression. If a character string or function, is should have the package name e.g., `"terra::rast"` or as an actual function, e.g., `base::readRDS`. In these cases, it will evaluate this function call while passing `targetFile` as the first argument. These will only work in the simplest of cases.

When more precision is required, the full call can be written and where the filename can be referred to as `targetFile` if the function is loading a file. If `preProcess` returns an object, `fun` should be set to `fun = NA`.

If there is a custom function call, is not in a package, `prepInputs` may not find it. In such cases, simply pass the function as a named argument (with same name as function) to `prepInputs`. See examples. NOTE: passing `fun = NA` will skip loading object into R. Note this will essentially replicate the functionality of simply calling `preProcess` directly.

`purge`

In options for control of purging the `CHECKSUMS.txt` file are:

`0` keep file

`1` delete file in `destinationPath`, all records of downloads need to be rebuilt

`2` delete entry with same `targetFile`

`4` delete entry with same `alsoExtract`

`3` delete entry with same `archive`

`5` delete entry with same `targetFile` & `alsoExtract`

`6` delete entry with same `targetFile`, `alsoExtract` & `archive`

`7` delete entry that same `targetFile`, `alsoExtract` & `archive` & `url`

will only remove entries in the `CHECKSUMS.txt` that are associated with `targetFile`, `alsoExtract` or `archive` When `prepInputs` is called, it will write or append to a (if already exists) `CHECKSUMS.txt` file. If the `CHECKSUMS.txt` is not correct, use this argument to remove it.

Author(s)

Eliot McIntire, Jean Marchal, and Tati Micheletti

See Also

postProcessTo(), downloadFile(), extractFromArchive(), postProcess().

Examples

```
if (requireNamespace("terra", quietly = TRUE) &&
  requireNamespace("withr", quietly = TRUE)) {
  library(reproducible)
  withr::local_dir(withr::local_tempdir())
  # Make a dummy study area map -- user would supply this normally
 coords <- structure(c(-122.9, -116.1, -99.2, -106, -122.9, 59.9, 65.7, 63.6, 54.8, 59.9),
    dim = c(5L, 2L)
  )
  studyArea <- terra::vect(coords, "polygons")
 terra::crs(studyArea) <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
  # Make dummy "large" map that must be cropped to the study area
  outerSA <- terra::buffer(studyArea, 50000)
 terra::crs(outerSA) <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
  tf <- normPath(file.path(tempdir2(), "prepInputs2.shp"))
  terra::writeVector(outerSA, tf)

  # run prepInputs -- load file, postProcess it to the studyArea

  studyArea2 <- prepInputs(
    targetFile = tf, to = studyArea,
    fun = "terra::vect",
    destinationPath = tempdir2()
  ) |>
    suppressWarnings() # not relevant warning here

  # clean up
  unlink("CHECKSUMS.txt")

  ########################################
  # Remote file using `url`
  ########################################
  if (identical(Sys.getenv("NOT_CRAN"), "true")) {
    data.table::setDTthreads(2)
    # download a zip file from internet, unzip all files, load as shapefile
    # Don't know all the files -- prepInputs will guess: if the downloaded file is an
    #   archive, extract all files, then if there is a .shp, load it with sf::st_read
    dPath <- file.path(tempdir2(), "ecozones")
    shpUrl <- paste0(
      "https://github.com/PredictiveEcology/reproducible/releases/download/v3.1.1/",
      "ecozone_shp.zip"
    )

    # Wrapped in a try because this needs internet
```

```
  shpEcozone <- try(prepInputs(destinationPath = dPath, url = shpUrl))

  ########################################
  # Local archive -- the same file, shipped with the package
  ########################################
  # The remaining examples use the copy in `inst/ex`, so they need no internet
  dPath <- file.path(tempdir2(), "ecozonesLocal")
  checkPath(dPath, create = TRUE)
  file.copy(system.file("ex/ecozone_shp.zip", package = "reproducible"), dPath)

  shpEcozone <- prepInputs(destinationPath = dPath, archive = "ecozone_shp.zip")

  # Robust to partial file deletions: the missing files are re-extracted
  unlink(dir(file.path(dPath, "Ecozones"), full.names = TRUE)[1:2])
  shpEcozone <- prepInputs(destinationPath = dPath, archive = "ecozone_shp.zip")

  # Once this is done, can be more precise in operational code:
  #  specify targetFile, alsoExtract, and fun, wrap with Cache
  ecozoneFilename <- file.path(dPath, "Ecozones", "ecozones.shp")
 # Note, you don't need to "alsoExtract" the archive... if the archive is not there, but the
  #   targetFile is there, it will not re-extract the archive.
  ecozoneFiles <- c("ecozones.dbf", "ecozones.prj", "ecozones.shp", "ecozones.shx")
  shpEcozone <- prepInputs(
    targetFile = ecozoneFilename,
    archive = "ecozone_shp.zip",
    fun = "terra::vect",
    alsoExtract = ecozoneFiles,
    destinationPath = dPath
  )

  # Add a study area to Crop and Mask to
  # Create a "study area"
 coords <- structure(c(-122.98, -116.1, -99.2, -106, -122.98, 59.9, 65.73, 63.58, 54.79, 59.9),
    dim = c(5L, 2L)
  )
  studyArea <- terra::vect(coords, "polygons")
 terra::crs(studyArea) <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"

  shpEcozoneSm <- Cache(prepInputs,
    archive = "ecozone_shp.zip",
    targetFile = reproducible::asPath(ecozoneFilename),
    alsoExtract = reproducible::asPath(ecozoneFiles),
    studyArea = studyArea,
    fun = "terra::vect",
    destinationPath = dPath,
    writeTo = "EcozoneFile.shp"
  ) # passed to determineFilename

  terra::plot(shpEcozone[, 1])
  terra::plot(shpEcozoneSm[, 1], add = TRUE, col = "red")
 }
 withr::deferred_run()
}
```

```
## Using quoted dlFun and fun -- this is not intended to be run but used as a template
## prepInputs(..., fun = customFun(x = targetFile), customFun = customFun)
##   # or more complex
##  test5 <- prepInputs(
##   targetFile = targetFileLuxRDS,
##   dlFun =
##    getDataFn(name = "GADM", country = "LUX", level = 0) # preProcess keeps file from this!
##   ,
##   fun = {
##     out <- readRDS(targetFile)
##     sf::st_as_sf(out)}
##  )
```

`prepInputsLog` *URL access log for* `prepInputs` / `preProcess`

Description

Controlled by `getOption("reproducible.urlLog")`. See the package option documentation for modes. `prepInputsLog()` returns the package-level in-memory records, which are populated in the default (`NULL`) and `TRUE` modes; `clearUrlLog()` empties them. Records written to an environment or function sink live there instead and are not retrievable through these accessors. Set the option to `FALSE` to disable logging entirely.

Usage

```
prepInputsLog()

clearUrlLog()
```

Value

`prepInputsLog()` returns a list of record lists. `clearUrlLog()` returns `NULL` invisibly.

Examples

```
# Every prepInputs()/preProcess() call that resolves a url records it, so a
# completed run can report exactly which data sources it pulled.
clearUrlLog()
prepInputsLog() # empty at the start of a session

# After any prepInputs()/preProcess() work, the log holds one record per url;
# rbindlist() turns it into a table for reporting.
# data.table::rbindlist(prepInputsLog())
```

`prepInputsWithTiles` *Alternative to* `prepInputs` *that can use Spatial Tiles stored locally or on Google Drive*

Description

[Experimental]

Downloads, processes and optionally uploads a `SpatRaster` object through a tiling intermediary. If the original `url` is for a very large object, but `to` is a relatively small subset of the area represented by the spatial file at `url`, then this function will potentially by-pass the download of the large file at `url` and instead only download the minimum number of tiles necessary to cover the `to` area. When `doUploads` is TRUE, then this function will potentially create and upload the tiles to `tileFolder`, prior to returning the spatial object, `postProcessed` to `to`. This function supports both Google Drive and HTTP(S) URLs.

Usage

```
prepInputsWithTiles(
  targetFile,
  url,
  destinationPath,
  to,
  tilesFolder = file.path(getOption("reproducible.inputPath"), "tiles"),
  urlTiles = getOption("reproducible.prepInputsUrlTiles", NULL),
  doUploads = getOption("reproducible.prepInputsDoUploads", FALSE),
  tileGrid = "CAN",
  numTiles = NULL,
  plot.grid = FALSE,
  purge = FALSE,
  verbose = getOption("reproducible.verbose"),
  ...
)
```

Arguments

`targetFile`	Character. Name of the target file to be downloaded or processed. If missing, it will be inferred from the URL or Google Drive metadata.
`url`	Character. URL to the full dataset (Google Drive or HTTP/S).
`destinationPath`	Character. Path to the directory where files will be downloaded and processed.
`to`	A spatial object (e.g., `SpatRaster`, `SpatVector`, `sf`, or `Spatial*`) defining the area of interest.
`tilesFolder`	A local file path to put tiles. If this is an absolute path, then that will be used; if it is a relative path, then it will be `file.path(destinationPath, tilesFolder)`
`urlTiles`	Character. URL to the tile source (e.g., Google Drive folder or HTTP/S endpoint). Default is `getOption("reproducible.prepInputsUrlTiles", NULL)`.

doUploads — Logical. Whether to upload processed tiles. Default is getOption("reproducible.prepInputsDoUploa FALSE).

tileGrid — Either length 3 character string, such as "CAN", to be sent to geodata::gadm(...) or an actual SpatVector object with a grid of polygons.

When a character code is used, the GADM boundaries must be downloaded, and geodata requires somewhere to put them. That location is resolved in this order, preferring somewhere persistent so the download happens only once:

1. geodata::geodata_path(), if the user has configured one;
2. getOption("reproducible.destinationPathShared") (or its deprecated alias reproducible.inputPaths), the option intended for large files reused across projects;
3. getOption("reproducible.inputPath"), the package default, which is under tempdir() and so is re-downloaded each session.

If the download cannot be completed – no path, server outage, no network – a warning is issued and a fixed Canada-wide extent is used instead, which may not match the area of interest.

numTiles — Integer. Number of tiles to generate. Optional.

plot.grid — Logical. Whether to plot the tile grid and area of interest. Default is FALSE.

purge — Logical or Integer. 0/FALSE (default) keeps existing CHECKSUMS.txt file and prepInputs will write or append to it. 1/TRUE will deleted the entire CHECKSUMS.txt file.

verbose — Logical or numeric. Controls verbosity of messages. Default is getOption("reproducible.verbose").

... — Either maskTo, cropTo (which will be used if to is not supplied, or arguments passed to writeRaster, e.g., datatype (used when writing tiles).

Details

This function can be triggered *inside* prepInputs if the to is supplied and both url and urlTiles are supplied. **NOTE**: urlTiles can be supplied using the option(reproducible.prepInputsUrlTiles = someGoogleDri so the original prepInputs function call can remain unaffected.

This function also uses a different checksumming procedure compared to the normal prepInputs. This function will assess the remote url for a hash. If that hash exists, then it will compare it to a local file with targetFile name, suffixed with .hash. If the two hashes differ (remote and local), then it will be redownloaded; otherwise the local one will be returned.

This function is useful for working with large spatial datasets, but where the user only requires a "relatively small" section of that dataset. This function will potentially bypass the full download and download only the tiles that are necessary for the to. It handles downloading only the required tiles based on spatial intersection with the target area, and supports resumable downloads from Google Drive or HTTP/S sources.

If targetFile is missing, the function attempts to infer it from the URL using the Content-Disposition header or the basename of the URL. For Google Drive URLs, it uses the file metadata.

Value

A single, merged SpatRaster object postProcessed to the area of interest (to), composed of the necessary tiles. If the post-processed file already exists locally, it will be returned directly.

See Also

googledrive::drive_get(), terra::rast(), terra::crop(), terra::merge()

Examples

```
if (FALSE) {
  to <- sf::st_as_sf(sf::st_sfc(sf::st_point(c(-123.3656, 48.4284)), crs = 4326))
  result <- prepInputsWithTiles(
    url = "https://example.com/data.tif",
    destinationPath = tempdir(),
    to = to,
    urlTiles = "https://example.com/tiles/",
    tileGrid = "CAN"
  )
}
```

---

prepopulateCacheAsync *Pre-populate the in-memory* showCache *cache for a given* cachePath

---

Description

[Experimental]

Forks a background process that runs showCache() against cachePath; subsequent showCache() / Cache()->showSimilar() calls in the same R session can then harvest the result instead of re-scanning the cache directory synchronously. Useful for very large caches (tens of thousands of entries) where the cold first scan can take a minute or more.

Idempotent: a second call with the same cachePath reuses the existing job. Skipped silently on Windows (forking-based) and when the parallel package isn't available.

This helper is called automatically the first time Cache() or showCache() is invoked against a given cachePath, so most users do not need to call it explicitly. It is exported for workflows that want to kick off the spawn early (e.g. inside setupProject()) so the fork has more wall-clock time to complete before the first manual showCache() call.

Usage

```
prepopulateCacheAsync(cachePath = getOption("reproducible.cachePath"))
```

Arguments

cachePath	A character path. Defaults to getOption("reproducible.cachePath").

Value

Invisibly returns the spawn job handle, or NULL if the spawn was skipped.

Examples

```
# Scanning a large flat-file cache is slow, so this warms it in the
# background; a later showCache()/Cache(showSimilar = TRUE) then returns
# without waiting. Guarded on interactive() because it spawns a process.
cachePath <- file.path(tempdir(), "prepopulateExample")
dir.create(cachePath, showWarnings = FALSE)

prepopulateCacheAsync(cachePath = cachePath)

unlink(cachePath, recursive = TRUE)
```

---

preProcessParams *Download, checksum, extract files*

Description

[Stable]

This does downloading (via `downloadFile`), checksumming (`Checksums`), and extracting from archives (`extractFromArchive`), plus cleaning up of input arguments (e.g., paths, function names). This is the first stage of three used in `prepInputs`.

Usage

```
preProcessParams(n = NULL)

preProcess(
  targetFile = NULL,
  url = NULL,
  archive = NULL,
  alsoExtract = NULL,
  destinationPath = getOption("reproducible.destinationPath", "."),
  fun = NULL,
  dlFun = NULL,
  quick = getOption("reproducible.quick"),
  overwrite = getOption("reproducible.overwrite", FALSE),
  purge = FALSE,
  verbose = getOption("reproducible.verbose", 1),
  .tempPath,
  .callingEnv = parent.frame(),
  ...
)
```

Arguments

`n`	Number of non-null arguments passed to `preProcess`. E.g., passing `n = 1` returns combinations with only a single non-NULL parameter. If `NULL` (default), all parameter combinations are returned.

`targetFile` Character string giving the filename (without relative or absolute path) to the eventual file (raster, shapefile, csv, etc.) after downloading and extracting from a zip or tar archive. This is the file *before* it is passed to `postProcess`. The internal checksumming does not checksum the file after it is `postProcessed` (e.g., cropped/reprojected/masked). Using `Cache` around `prepInputs` will do a sufficient job in these cases. See table in `preProcess()`.

`url` Optional character string indicating the URL to download from. If not specified, then no download will be attempted. If not entry exists in the `CHECKSUMS.txt` (in `destinationPath`), an entry will be created or appended to. This `CHECKSUMS.txt` entry will be used in subsequent calls to `prepInputs` or `preProcess`, comparing the file on hand with the ad hoc `CHECKSUMS.txt`. See table in `preProcess()`.

`archive` Optional character string giving the path of an archive containing `targetFile`, or a vector giving a set of nested archives (e.g., `c("xxx.tar", "inner.zip", "inner.rar")`). If there is/are (an) inner archive(s), but they are unknown, the function will try all until it finds the `targetFile`. See table in `preProcess()`. If it is `NA`, then it will *not* attempt to see it as an archive, even if it has archive-like file extension (e.g., `.zip`). This may be useful when an R function is expecting an archive directly.

`alsoExtract` Which files to fetch or extract in addition to `targetFile`. Spatial data rarely arrives as one file – a shapefile is five or six, and a raster often has a `.aux.xml`, `.ovr` or `.tfw` alongside carrying its attribute table, overviews or georeferencing – so this controls how much of that company comes along.

The default depends on where the files are coming from, because "everything" only means something when a container bounds it:

`alsoExtract`	**from an** `archive`	**from a** `url`
`NULL` (default)	every file in the archive	as for `"similar"`
`"similar"`	files sharing `targetFile`'s name without its extension	same, discovered by listing the remote directory
`NA` or `"none"`	`targetFile` only	`targetFile` only
a character vector	those files only	those files only

An archive bounds what "all files" can mean, so `NULL` extracts all of it. A `url` has no such bound – the server may hold thousands of unrelated files – so `NULL` there means `targetFile` and its companions, never the whole remote directory.

Each element of a character vector may also be a regular expression: if an element does not match any archive member literally (by relative path or basename), it is passed to `grep()` against the archive's file list and all matching members are extracted. For example, `alsoExtract = "CMD_sm|CMD_sp"` extracts every file whose name contains `CMD_sm` or `CMD_sp`. See table in `preProcess()`.

`destinationPath`

Character string of a directory in which to download and save the file that comes from `url` and is also where the function will look for `archive` or `targetFile`. To prevent repeated downloads in different locations, the user can also set `options("reproducible.dest` to one or more local file paths to search for the file before attempting to download. Default for that option is `NULL` meaning do not search locally. The previous name `options("reproducible.inputPaths")` is still accepted as a backwards-compatible alias.

fun	Optional. If specified, this will attempt to load whatever file was downloaded during preProcess via dlFun. This can be either a function (e.g., sf::st_read), character string (e.g., "base::load"), NA (for no loading, useful if dlFun already loaded the file) or if extra arguments are required in the function call, it must be a call naming targetFile (e.g., sf::st_read(targetFile, quiet = TRUE)) as the file path to the file to load. See details and examples below.
dlFun	Optional "download function" name, such as "raster::getData", which does custom downloading, in addition to loading into R. Still experimental.
quick	Logical. This is passed internally to Checksums() (the quickCheck argument), and to Cache() (the quick argument). This results in faster, though less robust checking of inputs. See the respective functions.
overwrite	Logical. Passed to writeTo (possibly inside postProcess) and postProcess.
purge	Logical or Integer. 0/FALSE (default) keeps existing CHECKSUMS.txt file and prepInputs will write or append to it. 1/TRUE will deleted the entire CHECKSUMS.txt file. Other options, see details.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
.tempPath	Optional temporary path for internal file intermediate steps. Will be cleared on.exit from this function.
.callingEnv	The environment where the function was called from. Used to find objects, if necessary.
...	Additional arguments passed to postProcess() and Cache(). Since ... is passed to postProcess(), these will ... will also be passed into the inner functions, e.g., cropInputs(). Possibly useful other arguments include dlFun which is passed to preProcess. See details and examples.

Value

A list with 5 elements: checkSums (the result of a Checksums after downloading), dots (cleaned up ..., including deprecated argument checks), fun (the function to be used to load the preProcessed object from disk), and targetFilePath (the fully qualified path to the targetFile).

Combinations of targetFile, url, archive, alsoExtract

Use preProcessParams() for a table describing various parameter combinations and their outcomes.

* If the url is a file on Google Drive, checksumming will work even without a targetFile specified because there is an initial attempt to get the remove file information (e.g., file name). With that, the connection between the url and the filename used in the 'CHECKSUMS.txt' file can be made.

Author(s)

Eliot McIntire

Examples

```
# preProcess() needs no url: a file already on disk is checksummed, placed in
# destinationPath, and (with fun = NA) reported rather than loaded.
dPath <- file.path(tempdir(), "preProcessExample")
shp <- system.file("ex/luxSmall/luxSmall.shp", package = "reproducible")

out <- preProcess(targetFile = shp, destinationPath = dPath, fun = NA)
basename(out$targetFilePath)

# A CHECKSUMS.txt is written, which is what makes a second call cheap.
dir(dPath)

unlink(dPath, recursive = TRUE)
```

`purgeChecksums` *Purge the checksums of a single file*

Description

This is a manual way of achieving `prepInputs(..., purge = 7)`, useful in cases where `prepInputs` is not called directly by the user, so it would be difficult to set `purge = 7`.

Usage

```
purgeChecksums(checksumFile, fileToRemove)
```

Arguments

`checksumFile`	A character string indicating the absolute path to the `CHECKSUMS.txt` file.
`fileToRemove`	The filename to remove from the `checksumFile`

Value

NULL. Run for its side effect, namely, and file removed from the '`CHECKSUMS.txt`' file.

Examples

```
# Set up a small CHECKSUMS.txt-like file, then purge one entry
tf <- tempfile(fileext = ".txt")
data.table::fwrite(
  data.table::data.table(file = c("a.tif", "b.tif"), checksum = c("x1", "x2")),
  tf
)
purgeChecksums(tf, "a.tif")
data.table::fread(tf)
unlink(tf)
```

rasterRead *A helper to* getOption("reproducible.rasterRead")

Description

A helper to getOption("reproducible.rasterRead")

Usage

```
rasterRead(...)
```

Arguments

...	Passed to the function parsed and evaluated from getOption("reproducible.rasterRead")

Value

A function, that will be the evaluated, parsed character string, e.g., eval(parse(text = "terra::rast"))

Examples

```
# returns the function named by getOption("reproducible.rasterRead")
rr <- getOption("reproducible.rasterRead")
rr
```

remapFilenames *Remap file names*

Description

Update file path metadata for file-backed objects (e.g., SpatRasters). Useful when moving saved objects between projects or machines.

Usage

```
remapFilenames(obj, tags, cachePath = getOption("reproducible.cachePath"), ...)
```

Arguments

obj	(optional) object whose file path metadata will be remapped
tags	cache tags data.table object
cachePath	character string specifying the path to the cache directory or NULL
...	Additional path arguments, passed to absoluteBase() and modifyListPaths()

Examples

```
# A file-backed object cached on one machine records where its files lived.
# remapFilenames() translates those to where they belong on THIS machine,
# using the anchors (e.g. inputPath) supplied in `paths`.
inputDir <- file.path(tempdir(), "remapExample")
dir.create(inputDir, showWarnings = FALSE)

tags <- c(
  "origFilename:dem.tif",
  "origRelName:dem.tif",
  "relToWhere:inputPath", # the anchor the file was stored relative to
  "filesToLoad:/some/other/machine/inputs/dem.tif"
)

newFiles <- remapFilenames(
  tags = tags, cachePath = NULL,
  paths = list(inputPath = inputDir)
)
newFiles$newName # now under THIS machine's inputPath

unlink(inputDir, recursive = TRUE)
```

---

reproducibleOptions reproducible *options*

Description

These provide top-level, powerful settings for a comprehensive reproducible workflow. To see defaults, run `reproducibleOptions()`. See Details below.

Usage

```
reproducibleOptions()
```

Details

Below are options that can be set with `options("reproducible.xxx" = newValue)`, where `xxx` is one of the values below, and `newValue` is a new value to give the option. Sometimes these options can be placed in the user's `.Rprofile` file so they persist between sessions.

The following options are likely of interest to most users:

`ask` Default: `TRUE`. Used in `clearCache()` and `keepCache()`.

`cacheChaining` Default: `FALSE`. Used in `Cache()` in the `.cacheChaining` argument.

`cachePath` Default: `NULL`. Used in `Cache()` and many others. The option is no longer pre-set when the package is loaded; instead it is resolved lazily on first use by an entry point (`Cache()`, `clearCache()`, `showCache()`, `keepCache()`, ...). If still unset at that point, it is set to `.reproducibleTempCacheDir()` for the rest of the session. This lets project-setup layers

(e.g. SpaDES.project::setupProject()) detect "unset" cleanly and avoids committing every R session to a session-tempdir path that would not persist across sessions. Set this early (e.g. in your project setup script) to use a persistent cache.

cacheSaveFormat Default: "rds". What save format to use; currently, "qs" (which will use qs2 package as of reproducible version ">= 2.1.3"), "qs2", or "rds".

cacheSpeed Default "slow". One of "slow" or "fast" (1 or 2). "slow" uses digest::digest internally, which is transferable across operating systems, but much slower than digest::digest(algo = "spooky). So, if all caching is happening on a single machine, "fast" would be a good setting.

checkRemoteHash Default: FALSE. Used in preProcess() / prepInputs(). Controls whether pp_remote_hash_check re-contacts the remote source (e.g. Google Drive, HTTP HEAD) when a .hash sidecar from a previous successful match already exists in destinationPath. With the default (FALSE), the sidecar is trusted and the remote check is skipped — typically saving 1–2 s per file when the cluster cache is warm. Set to TRUE to force a remote round-trip on every call (the pre-3.0.0.9050 behaviour); use this if the upstream file may change and you need to detect that. Removing the <file>_*.hash sidecar also forces a re-check.

conn Default: NULL. Sets a specific connection to a database, e.g., dbConnect(drv = RSQLite::SQLite()) or dbConnect(drv = RPostgres::Postgres(). For remote database servers, setting one connection may be far faster than using drv which must make a new connection every time.

destinationPath Default: NULL. Used in prepInputs() and preProcess(). Can be set globally here.

downloadProgressInterval Default: 2, in seconds. Minimum interval between download-progress lines in non-interactive / non-dynamic sessions (logged runs, CI, a SpaDES simInit). In those sessions httr2's native cli progress bar emits nothing, so preProcess() instead streams the body and reports progress via messagePreProcess() (which the calling app can timestamp). In a dynamic terminal the native in-place bar is used and this has no effect.

drv Default: RSQLite::SQLite(). Sets the default driver for the backend database system. Only tested with RSQLite::SQLite() and RPostgres::Postgres().

dryRun Default: FALSE.

fileBackedAnchors Default: NULL. A named list of "anchor" directories (e.g. the result of SpaDES paths(sim): cachePath, inputPath, outputPath, modulePath, ...) used to make *file-backed* objects (such as a terra SpatRaster) portable across machines and users. A file-backed object embeds an *absolute* path to its backing file; when stored relative to a named anchor here, Cache records the anchor name plus the path relative to it, and on load rebuilds the file under the *receiver's* anchor of the same name. Each entry may hold one or more directories, and the most specific (longest) matching anchor wins. The same named list must be set on both the machine that writes the cache entry and the one that reads it (e.g. a shared cloud cache). When the file lives under no anchor, or the anchor name is not set on the receiver, the object is restored *self-contained under the receiver's* cachePath rather than the producing machine's absolute path. cachePath and the current working directory are always available as fallback anchors.

futurePlan **No effect as of version 3.2.0.** Default: FALSE. This previously enabled future-backed background downloading and cache writing on Linux. Those paths were off by default, effectively untested, and have been removed; the option is retained only because downstream packages set it, and setting it now does nothing.

gdalwarp **Deprecated — do not use.** Default: FALSE. This option previously switched postProcessTo to use sf::gdal_utils("warp") for a specific combination of raster/vector inputs. It is no

longer needed: current versions of terra handle this case well and produce equivalent results without the GDAL detour. The option is retained only for backwards compatibility and will be removed in a future release.

gdalwarpThreads **Deprecated — do not use** (see gdalwarp above). Default: 2. Previously set -wo NUM_THREADS= for gdalProject.

gdriveNoAuth Default: FALSE. Used in prepInputs() and preProcess(). When TRUE, Google Drive files are accessed without a googledrive token (i.e. no drive_auth()): both the *metadata* read in assessGoogle() and the download itself are done anonymously (the metadata read by deauthorizing googledrive so it uses an API key). This only works for files shared as "Anyone with the link". The same no-auth path is taken automatically — regardless of this option — when no Drive token is loaded (the common "cloud reader" case) or when the supplied url is the public web-download form, e.g. https://drive.google.com/uc?export=download&id=<ID>. A loaded token (a "cloud writer" who needs auth to write a shared cloud cache) is left intact.

destinationPathShared Default: NULL. Used in prepInputs() and preProcess(). If set to a path, this will cause these functions to save their downloaded and preprocessed file to this location, with a hardlink (via file.link) to the file created in the destinationPath. This can be used so that individual projects that use common data sets can maintain modularity (by placing downloaded objects in their destinationPath, but also minimize re-downloading the same (perhaps large) file over and over for each project. Because the files are hardlinks, there is no extra space taken up by the apparently duplicated files.

Note: the previous name for this option was `reproducible.inputPaths`; the old name is still accepted and will continue to work, but `reproducible.destinationPathShared` is preferred going forward (it matches the [prepInputs()] naming family).

destinationPathSharedRecursive Default: FALSE. Used in prepInputs() and preProcess(). Should reproducible.destinationPathShared be searched recursively for existence of a file?

Note: the previous name for this option was `reproducible.inputPathsRecursive`; the old name is still accepted but `reproducible.destinationPathSharedRecursive` is preferred.

inputPaths **Deprecated** — use reproducible.destinationPathShared instead. Retained for backwards compatibility; if set and reproducible.destinationPathShared is NULL, the value of reproducible.inputPaths is used automatically.

inputPathsRecursive **Deprecated** — use reproducible.destinationPathSharedRecursive instead. Retained for backwards compatibility.

leaveOnDisk Default: TRUE. Used in postProcess(). When there is a SpatRaster object, should postProcess force any file-backed object, to use the file-based, memory-safe tools within terra (by temporarily setting terraOption(memfrac = 0). Alternatively, if this is set to FALSE, then postProcess will let terra decide on its own based on its internal cues (largely based on memfrac, maxmem terraOptions). This will be ignored, however, if the user has set the terraOptions away from its default of 0.5. The default increases predictability of whether the returned object is on disk or in memory.

terraMemmax Default: 2 (gigabytes). Used in postProcessTo(). Caps terra's per-raster memory budget for the duration of a postProcessTo() call by temporarily setting terraOptions(memmax = ...), restored via on.exit(). Small values force terra to process in chunks, which on high-RAM machines is substantially faster than letting it pull whole rasters into RAM (in

benchmarks on a 1TB-RAM machine, `memmax = 4` was ~45% faster and used ~3x less peak RSS than the unbounded default; the 2GB default is conservative for shared nodes). Set to `NULL` to disable and let `terra` choose. **Respects user-set values**: if the caller has already set `terraOptions(memmax = ...)` to a positive finite value, `postProcessTo()` leaves it alone – the option only applies when `terra`'s `memmax` is at its default ("ignored": `NA`, `NULL`, or `<= 0`; terra's out-of-the-box default is `-1`). See also `memfrac` in `terra::terraOptions()`; a `memfrac` ceiling of `0.1` is sensible on shared machines.

`memoisePersist` Default: `FALSE`. Used in `Cache()`. Should the memoised copy of the Cache objects persist even if `reproducible` reloads e.g., via `devtools::load_all`? This is mostly useful for developers of `reproducible`. If `TRUE`, a object named `paste0(".reproducibleMemoise_", cachePath)` will be placed in the `.GlobalEnv`, i.e., one for each `cachePath`.

`nThreads` Default: `1`. The number of threads to use for reading/writing cache files.

`objSize` Default: `TRUE`. Logical. If `TRUE`, then object sizes will be included in the cache database. Simplying calculating object size of large objects can be time consuming, so setting this to `FALSE` will make caching up to 10% faster, depending on the objects.

`overwrite` Default: `FALSE`. Used in `prepInputs()`, `preProcess()`, `downloadFile()`, and `postProcess()`.

`parallel.cores` Default: `NULL`, meaning `parallelly::freeCores()`. Degree of parallelism for **CPU-bound** work, i.e. tiling in `prepInputs()` when `numTiles` is supplied. Being CPU work, it is additionally capped by `mc.cores`. Setting it to `1L` also disables the background `showCache()` pre-warm fork, which costs one extra process (see `showCachePreWarm` below).

`parallel.download` Default: `NULL`, meaning `48L`. Number of concurrent ranged connections used to fetch a single large file (equivalently, the number of byte-range parts it is split into). Deliberately **not** derived from the core count: network throughput is bounded by bandwidth and per-connection server shaping, not by CPUs, so this is not capped by `mc.cores` either. Set `1L` to force the single-stream path. As with the deprecated options it replaces, this has no effect unless `reproducible.urlRemap` is set.

`parallel.upload` Default: `NULL`, meaning `7L`. Number of concurrent uploads to Google Drive. Network-bound like `parallel.download`, so likewise not core-derived and not capped by `mc.cores`; more than about 7 tends to be slower, depending on connection speed.

`parallel.streams` **[Deprecated]**)Useparallel.downloadinstead, which takes precedence when both are set. Still producible.urlRemap** (see below): if no remap is set, downloads are always single-stream, regardless Remaphook actually redirected** to a (Range-capable) mirror – *not* for arbitrary range-capable or Ranges: bytesand the file is larger thanreproducible.parallel.threshold; otherwise, and on any failure, it producible.parallel.minConcurrentFrac– and also falls back.) Splitting into many small parts keeps retri
Use `parallel.download` instead, which takes precedence when both are set. Still honoured when it alone is set. Previously defaulted to `parallelly::availableCores() - 1`, which meant a user setting `mc.cores` to limit forking silently throttled downloads. The maximum number of ranged parts that download **simultaneously**; the rest queue until a connection frees up. This bounds the burst of concurrent TLS handshakes, which some stacks (notably Windows) refuse when all `reproducible.parallel.streams` are opened at once — the symptom being most parts failing immediately at connection time. Set a positive integer to override the default ceiling.

`parallel.connecttimeout` Default: `30L`, in seconds. The per-connection establishment timeout for each ranged stream. This is distinct from `reproducible.timeout` (the overall download timeout, which may be hours): a short, dedicated cap so a stalled handshake fails its own part quickly and is retried rather than hanging.

`parallel.minConcurrentFrac` Default: `0.25`. A guard against a redirected mirror that advertises Range support but actually caps concurrent connections per IP. If the *first* concurrent attempt completes fewer than this fraction of the parts (most having stalled at 0 bytes), the parallel path gives up immediately and falls back to a single stream, instead of grinding through every retry. Set to `0` to disable the early fallback.

`parallel.threshold` Default: `10 * 1024^2` (10 MiB), in bytes. Files at or below this size are always downloaded single-stream; only files larger than this use the parallel ranged path. Like `reproducible.parallel.streams`, this has no effect unless you have opted in via `reproducible.urlRemap`.

`quick` Default: `FALSE`. Used in `Cache()`. This will cause `Cache` to use `file.size(file)` instead of the `digest::digest(file)`. Less robust to changes, but faster. *NOTE: this will only affect objects on disk.*

`rasterRead` Used during `prepInputs` when reading `.tif`, `.grd`, and `.asc` files. Default: `terra::rast`. Can be `raster::raster` for backwards compatibility. Can be set using environment variable `R_REPRODUCIBLE_RASTER_READ`.

`shapefileRead` Default `NULL`. Used during `prepInputs` when reading a `.shp` file. If `NULL`, it will use `sf::st_read` if `sf` package is available; otherwise, it will use `raster::shapefile`

`showSimilar` Default `FALSE`. Passed to `Cache`.

`showCachePreWarm` Default `TRUE` (override with environment variable `R_REPRODUCIBLE_SHOWCACHE_PREWARM`). When `TRUE`, a `Cache(showSimilar = TRUE)` call spawns a one-time background process (a fork; not on Windows) that pre-scans the flat-file cache so the subsequent `showCache()`/`showSimilar` lookup returns quickly for large caches. This is skipped automatically under a DBI backend (`useDBI(TRUE)`), which is answered from an index. **Advanced option:** set to `FALSE` to disable the automatic pre-warm entirely – useful in memory-constrained runners that touch many distinct `cachePaths` in one session (e.g. `covr::package_coverage()`), where the per-path forks can accumulate. Explicit `prepopulateCacheAsync()` calls are unaffected.

`preDigestDump` Default: `NULL` (off). A diagnostic for "why is my `cacheId` different on this machine than that one?" (e.g. a cloud cache that will not share across OSs). Unlike `showSimilar` (closest prior call only) or `dryRun`/`verbose`, this dumps the **full** element-by-element `name = hash` list that produced the `cacheId`, for *every* `Cache()` call (including ones built inside other packages, e.g. SpaDES.core events). Set to `TRUE` to print each call's sorted list via `messageCache()`, or to a **directory path** to write one `preDigest_<functionName>[_<n>].txt` file per call. Point it at a fresh, empty directory on each machine, run, then `diff` the two directories: the differing `name = hash` line is exactly what is splitting the `cacheId`.

`preDigestDumpPattern` Default: `NULL`. Optional regular expression matched against a call's `.functionName`; when set, only matching `Cache()` calls are dumped by `reproducible.preDigestDump` (e.g. `"init|inputObjects"`).

`testCharacterAsFile` Default `FALSE`. The behaviour of `.robustDigest` on `character` vectors prior to `reproducible == 2.1.2` was that the function would test for whether they were filenames by using `file.exists`. If it was a filename, then it would digest the file content. In cases of a character vector or a data.frame of "filenames", this could cause long hanging of the R system as it tries to digest the file contents of potentially many files. This behaviour is not transparent to a user. Now the default is to **not** digest the file content of a `character` vector even if they are filenames. To force file content digesting, then convert to either `asPath` or `fs::as_fs_path`. Or set this option to `TRUE` and the previous behaviour will return, where it tries to guess whether a character vector is filenames or not, and if it is, then digest the file content.

`connecttimeout` Default: `30L`, in seconds. The per-connection establishment (TLS handshake) timeout for the single-stream download in `preProcess`. This is distinct from `reproducible.timeout` (the overall download budget, which may be hours): a short, dedicated cap so a stalled or flaky connect fails quickly and is retried rather than hanging for the full timeout. Mirrors `reproducible.parallel.connecttimeout` for the parallel ranged path.

`timeout` Default `12000`. Used in `preProcess` when downloading occurs. If a user has `R.utils` package installed, `R.utils::withTimeout( , timeout = getOption("reproducible.timeout"))` will be wrapped around the download so that it will timeout (and error) after this many seconds.

`urlLog` Default: `NULL`. Controls whether `prepInputs()` / `preProcess()` keep a record of the files and web addresses (URLs) they download. `NULL` (the default) records each download as a permanent tag on the matching cache entry, which you can look up later with `showCache(userTags = "reproducible.url")`; it keeps no in-session list. `TRUE` additionally keeps an in-memory list for the current session, which you can read with `prepInputsLog()` and empty with `clearUrlLog()`. `FALSE` turns the recording off completely. Advanced: you may instead supply an environment (records are appended to `env$records`, which you own and manage) or a function (called once with each record).

`useCache` Default: `TRUE`. Used in `Cache()`. If `FALSE`, then the entire `Cache` machinery is skipped and the functions are run as if there was no Cache occurring. Can also take 2 other values: `'overwrite'` and `'devMode'`. `'overwrite'` will cause no recovery of objects from the cache repository, only new ones will be created. If the hash is identical to a previous one, then this will overwrite the previous one. `'devMode'` will function as normally `Cache` except it will use the `userTags` to determine if a previous function has been run. If the `userTags` are identical, but the digest value is different, the old value will be deleted from the cache repository and this new value will be added. This addresses a common situation during the development stage: functions are changing frequently, so any entry in the cache repository will be stale following changes to functions, i.e., they will likely never be relevant again. This will therefore keep the cache repository clean of stale objects. If there is ambiguity in the `userTags`, i.e., they do not uniquely identify a single entry in the `cachePath`, then this option will default back to the non-dev-mode behaviour to avoid deleting objects. This, therefore, is most useful if the user is using unique values for `userTags`.

`urlRemap` Default: `NULL` (feature off). **This is the opt-in switch for the faster download path.** It may be set to any of: a function `function(url, filename)`; a manifest `data.frame` with `filename` and `url` columns; or a length-one character path/URL to a CSV with those columns. For the `data.frame`/CSV forms, `reproducible` builds the remap function internally (once, then cached) — so a novice can simply write `options(reproducible.urlRemap = read.csv("manifest.csv"))` without calling `makeUrlRemap()` themselves. However supplied, it is consulted in the download path once the target `filename` has been resolved (for Google Drive URLs, after the `drive_get()` lookup). The function may return an alternative URL to download from instead, e.g. a public mirror that supports HTTP Range requests (which then triggers the parallel ranged download governed by `reproducible.parallel.streams` and `reproducible.parallel.threshold`); returning `NULL` or the original URL keeps the behaviour unchanged. A function that errors is ignored (with a warning) so a broken remap cannot break a download. With the default `NULL`, no remapping occurs and downloads behave exactly as before.

`reproducible.useCacheV3` Default: `TRUE`. If this is set to `FALSE`, it will use the old `Cache` source code. This will only be available for a short period before it is deleted from the package. See

also reproducible.digestVersion. It is not guaranteed to be identical to using a previous version of reproducible (<3.0).

useCloud Default FALSE. Passed to Cache.

useDBI Default: FALSE, i.e., the file-backed cache metadata backend, which writes one small metadata file per cacheId alongside the cached object in cacheOutputs/. This needs no database packages, works on network filesystems (e.g., NFS, CIFS), where SQLite file locking is unreliable, and makes cloud caching straightforward because each entry's metadata is a self-contained, uploadable file. If TRUE, cache metadata are instead kept in a **DBI** database – SQLite (cache.db) by default, or any **DBI** backend supplied via reproducible.drv/reproducible.conn. This answers showCache() from the database rather than by scanning the cache directory, which is faster on large cache repositories. It requires both **DBI** and **RSQLite**; if either is missing, the option silently reverts to FALSE with a message. Switching this option on an existing cache repository is supported: the metadata are converted to the other backend on first use, without loss. Default value can be overridden by setting environment variable R_REPRODUCIBLE_USE_DBI to "true" or "false". As of version 0.3, the database backend is **DBI** instead of **archivist**.

useGdown Default: FALSE. If a user provides a Google Drive url to preProcess/prepInputs, reproducible will use the googledrive package. This works reliably in most cases. However, for large files on unstable internet connections, it will stall and stop the download with no error. If a user is finding this behaviour, they can install the gdown package, making sure it is available on the PATH. This call to gdown will only work for files that do not need authentication. If authentication is needed, dlGoogle will fall back to googledrive::drive_download, even if this option is TRUE, with a message. .

useMemoise Default: FALSE. Used in Cache(). If TRUE, recovery of cached elements from the cachePath will use memoise::memoise. This means that the 2nd time running a function will be much faster than the first in a session (which either will create a new cache entry to disk or read a cached entry from disk). *NOTE: memoised values are removed when the R session is restarted.* **This option will use more RAM** and so may need to be turned off if RAM is limiting. clearCache of any sort will cause all memoising to be 'forgotten' (memoise::forget).

useNewDigestAlgorithm Default: 1. Option 1 is the version that has existed for sometime. There is now an option 2 which is substantially faster. It will, however, create Caches that are not compatible with previous ones. Options 1 and 2 are not compatible with the earlier 0. 1 and 2 will make Cache less sensitive to minor but irrelevant changes (like changing the order of arguments) and will work successfully across operating systems (especially relevant for the new cloudCache function.

useTerra Default: FALSE. The GIS operations in postProcess, by default use primarily the Raster package. The newer terra package does similar operations, but usually faster. A user can now set this option to TRUE and prepInputs and several components of postProcess will use terra internally.

verbose Default: FALSE. If set to TRUE then every Cache call will show a summary of the objects being cached, their object.size and the time it took to digest them and also the time it took to run the call and save the call to the cache repository or load the cached copy from the repository. This may help diagnosing some problems that may occur.

digestVersion Default: NULL (which resolves to 4). A single integer that selects the digest (i.e. cacheId) algorithm used by Cache(). This is the going-forward control; it supersedes the booleans digestV3 and digestV4 below (which are still honoured when digestVersion

is unset). Higher numbers are newer; each builds on the previous one. (Note: this is the *digest-algorithm* version, **not** the version of the `reproducible` **package** — although the lower numbers echo the package generation that introduced them. References below to e.g. "the `reproducible` package v3.x" mean the package release, not this option.) The versions:

- `2` *Some of* the hash assembly used by older `reproducible` **package** versions (before package v3.0.0), for partial backwards compatibility with caches they created. It cannot be made exact for all classes, particularly file-backed objects. (Equivalent to the old `digestV3 = FALSE`.)
- `3` The hash assembly introduced in the `reproducible` **package** v3.x (the default through package v3.1.1): includes the names of list elements and several other tweaks. `sf`/`SpatVector` use the original (pre-platform-stable) geometry digest, which could differ across operating systems. Set `digestVersion = 3` to reproduce the `cacheId`s of the `reproducible` package v3.1.1 and earlier, and to avoid the one-time cache invalidation described under `4`.
- `4` **Default.** As `3`, but `sf` and `SpatVector` objects are digested with a platform-stable algorithm: the geometry is the numeric vertex matrix with coordinates rounded to a fixed precision (plus the geometry type), and attributes are kept in feature order with columns sorted by a locale-independent method. The same vector data then produces the same `cacheId` on Windows, macOS and Linux (digest version `3` could differ across operating systems, preventing shared/cloud caching of these objects), and an `sf` object and its `SpatVector` equivalent digest identically. Requires the **terra** package. **NOTE:** because `4` is now the default, the `cacheId` of every `sf`/`SpatVector` object differs from that produced by the `reproducible` package v3.1.1 and earlier, so cached results that involved such objects are *recomputed once* under the new algorithm. Set `digestVersion = 3` to avoid this.

`digestV3` **Superseded by** `digestVersion` (still honoured when `digestVersion` is unset). Default: `TRUE`. `TRUE` leaves the version at its default; `FALSE` is equivalent to `digestVersion = 2` (see above).

`digestV4` **Superseded by** `digestVersion` (still honoured when `digestVersion` is unset). Default: `FALSE`. `TRUE` is equivalent to `digestVersion = 4`, which is now also the default (see above).

Value

This function returns a list of all the options that the `reproducible` package sets and uses. See below for details of each.

Advanced

The following options are likely not needed by a user.

`cloudChecksumsFilename` Default: `file.path(dirname(.reproducibleTempCacheDir()), "checksums.rds")`. Used as an experimental argument in `Cache()`

`length` Default: `Inf`. Used in `Cache()`, specifically to the internal calls to `CacheDigest()`. This is passed to `digest::digest`. Mostly this would be changed from default `Inf` if the digesting is taking too long. Use this with caution, as some objects will have *many* `NA` values in their first *many* elements

useragent Default: "https://github.com/PredictiveEcology/reproducible". User agent for downloads using this package.

Examples

```
# See all reproducible options and their defaults
str(reproducibleOptions(), max.level = 1, list.len = 5)
```

---

retry *A wrapper around* try *that retries on failure*

Description

This is useful for functions that are "flaky", such as curl, which may fail for unknown reasons that do not persist.

Usage

```
retry(
  expr,
  envir = parent.frame(),
  retries = 5,
  exponentialDecayBase = 1.3,
  silent = TRUE,
  exprBetween = NULL,
  messageFn = message
)
```

Arguments

expr	An expression to run, i.e., rnorm(1), similar to what is passed to try
envir	The environment in which to evaluate the quoted expression, default to parent.frame(1)
retries	Numeric. The maximum number of retries.
exponentialDecayBase	Numeric > 1.0. The delay between successive retries will be runif(1, min = 0, max = exponentialDecayBase ^ i - 1) where i is the retry number (i.e., follows seq_len(retries))
silent	Logical indicating whether to try silently.
exprBetween	Another expression that should be run after a failed attempt of the expr. This should return a named list, where the names indicate the object names to update in the main expr, and the return value is the new value. (previous versions allowed a non-list return, but where the final line had to be an assignment operator, specifying what object (that is used in expr) will be updated prior to running the expr again. For backwards compatibility, this still works).
messageFn	A function for messaging to console. Defaults to message

Details

Based on https://github.com/jennybc/googlesheets/issues/219#issuecomment-195218525.

Value

As with `try`, so the successfully returned `return()` from the `expr` or a `try-error`.

Examples

```
# retries a flaky expression; here it simply succeeds on the first try
retry(rnorm(1), retries = 3)
```

---

saveToCache *Save an object to Cache*

Description

This is not expected to be used by a user as it requires that the `cacheId` be calculated in exactly the same as it calculated inside `Cache` (which requires `match.call` to match arguments with their names, among other things).

Usage

```
saveToCache(
  cachePath = getOption("reproducible.cachePath"),
  cacheSaveFormat = getOption("reproducible.cacheSaveFormat"),
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  obj,
  userTags,
  cacheId,
  linkToCacheId = NULL,
  verbose = getOption("reproducible.verbose")
)
```

Arguments

`cachePath`	A repository used for storing cached objects. This is optional if `Cache` is used inside a SpaDES module.
`cacheSaveFormat`	Character string: currently either `qs` or `rds`. Defaults to `getOption("reproducible.cacheSaveFormat"` `qs` may be faster but appears to have narrower range of conditions that work; `rds` is safer, and may be slower.
`drv`	If using a database backend, `drv` must be an object that inherits from `DBIDriver` (e.g., `RSQLite::SQLite`).

conn	an optional DBIConnection object, as returned by dbConnect().
obj	The R object to save to the cache
userTags	A character vector with descriptions of the Cache function call. These will be added to the Cache so that this entry in the Cache can be found using userTags e.g., via showCache().
cacheId	The hash string representing the result of .robustDigest
linkToCacheId	Optional. If a cacheId is provided here, then a file.link will be made to the file with that cacheId name in the cache repo. This is used when identical outputs exist in the cache. This will save disk space.
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t

Value

This is used for its side effects, namely, it will add the object to the cache and cache database.

---

searchFull *Search up the full scope for functions*

Description

This is like base::search but when used inside a function, it will show the full scope (see figure in the section *Binding environments* on http://adv-r.had.co.nz/Environments.html). This full search path will be potentially much longer than just search() (which always starts at .GlobalEnv).

searchFullEx shows an example function that is inside this package whose only function is to show the Scope of a package function.

Usage

```
searchFull(env = parent.frame(), simplify = TRUE)

searchFullEx()
```

Arguments

env	The environment to start searching at. Default is calling environment, i.e., parent.frame()
simplify	Logical. Should the output be simplified to character, if possible (usually it is not possible because environments don't always coerce correctly)

Details

searchFullEx can be used to show an example of the use of searchFull.

Value

A list of environments that is the actual search path, unlike `search()` which only prints from `.GlobalEnv` up to `base` through user attached packages.

See Also

`base::search()`

Examples

```
seeScope <- function() {
  searchFull()
}
seeScope()
searchFull()
searchFullEx()
```

---

set.randomseed *Set seed with a random value using Sys.time()*

Description

This will set a random seed.

Usage

```
set.randomseed(set.seed = TRUE)
```

Arguments

`set.seed`	Logical. If `TRUE`, the default, then the function will call `set.seed` internally with the new random seed.

Details

This function uses 6 decimal places of `Sys.time()`, i.e., microseconds. Due to integer limits, it also truncates at 1000 seconds, so there is a possibility that this will be non-unique after 1000 seconds (at the microsecond level). In tests, this showed no duplicates after 1e7 draws in a loop, as expected.

Value

This will return the new seed invisibly. However, this is also called for its side effects, which is a new seed set using `set.seed`

Note

This function does not appear to be as reliable on R <= 4.1.3

showCache *Examining and modifying the cache*

Description

These are convenience wrappers around DBI package functions. They allow the user a bit of control over what is being cached.

Usage

```
clearCache(
  x,
  userTags = character(),
  after = NULL,
  before = NULL,
  fun = NULL,
  cacheId = NULL,
  ask = getOption("reproducible.ask"),
  useCloud = FALSE,
  cloudFolderID = getOption("reproducible.cloudFolderID", NULL),
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  ...
)

## S4 method for signature 'ANY'
clearCache(
  x,
  userTags = character(),
  after = NULL,
  before = NULL,
  fun = NULL,
  cacheId = NULL,
  ask = getOption("reproducible.ask"),
  useCloud = FALSE,
  cloudFolderID = getOption("reproducible.cloudFolderID", NULL),
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  ...
)

cc(secs, ..., verbose = getOption("reproducible.verbose"))

showCache(
  x,
```

```
  userTags = character(),
  after = NULL,
  before = NULL,
  fun = NULL,
  cacheId = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  ...
)

## S4 method for signature 'ANY'
showCache(
  x,
  userTags = character(),
  after = NULL,
  before = NULL,
  fun = NULL,
  cacheId = NULL,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  ...
)

keepCache(
  x,
  userTags = character(),
  after = NULL,
  before = NULL,
  ask = getOption("reproducible.ask"),
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  ...
)

## S4 method for signature 'ANY'
keepCache(
  x,
  userTags = character(),
  after = NULL,
  before = NULL,
  ask = getOption("reproducible.ask"),
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  verbose = getOption("reproducible.verbose"),
  ...
```

)

Arguments

x	A simList or a directory containing a valid Cache repository. Note: For compatibility with Cache argument, cachePath can also be used instead of x, though x will take precedence.
userTags	Character vector. If used, this will be used in place of the after and before. Specifying one or more userTag here will clear all objects that match those tags. Matching is via regular expression, meaning partial matches will work unless strict beginning (^) and end ($) of string characters are used. Matching will be against any of the 3 columns returned by showCache(), i.e., artifact, tagValue or tagName. Also, if length(userTags) > 1, then matching is by and. For or matching, use | in a single character string. See examples.
after	A time (POSIX, character understandable by data.table). Objects cached after this time will be shown or deleted.
before	A time (POSIX, character understandable by data.table). Objects cached before this time will be shown or deleted.
fun	An optional character vector describing the function name to extract. Only functions with this/these functions will be returned.
cacheId	An optional character vector describing the cacheIds to extract. Only entries with this/these cacheIds will be returned. If useDBI(FALSE), this will also be dramatically faster than using userTags, for a large cache.
ask	Logical. If FALSE, then it will not ask to confirm deletions using clearCache or keepCache. Default is TRUE
useCloud	Logical. If TRUE, then every object that is deleted locally will also be deleted in the cloudFolderID, if it is non-NULL
cloudFolderID	A googledrive dribble of a folder, e.g., using drive_mkdir(). If left as NULL, the function will create a cloud folder with name from last two folder levels of the cachePath path, : paste0(basename(dirname(cachePath)), "_", basename(cachePath)). This cloudFolderID will be added to options("reproducible.cloudFolder but this will not persist across sessions. If this is a character string, it will treat this as a folder name to create or use on GoogleDrive.
drv	If using a database backend, drv must be an object that inherits from DBIDriver (e.g., RSQLite::SQLite).
conn	an optional DBIConnection object, as returned by dbConnect().
verbose	Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., options('reproducible.verbose' = 0) to reduce t
...	Other arguments. Can be in the form of tagKey = tagValue, such as, class = "numeric" to find all entries that are numerics in the cache. Note: the special cases of cacheId and fun have their own named arguments in these functions. Also can be regexp = xx, where xx is TRUE if the user is passing a regular expression. Otherwise, userTags will need to be exact matches. Default is missing, which is the same as TRUE. If there are errors due to regular expression

problem, try FALSE. For cc, it is passed to clearCache, e.g., ask, userTags. For showCache, it can also be sorted = FALSE to return the object unsorted.

secs — Currently 3 options: the number of seconds to pass to clearCache(after = secs), a POSIXct time e.g., from Sys.time(), or missing. If missing, the default, then it will delete the most recent entry in the Cache.

Details

If neither after or before are provided, nor userTags, then all objects will be removed. If both after and before are specified, then all objects between after and before will be deleted. If userTags is used, this will override after or before.

cc(secs) is just a shortcut for clearCache(repo = currentRepo, after = secs), i.e., to remove any cache entries touched in the last secs seconds. Since, secs can be missing, this is also be a shorthand for "remove most recent entry from the cache".

clearCache remove items from the cache based on their userTag or times values.

keepCache remove all cached items *except* those based on certain userTags or times values.

showCache display the contents of the cache.

By default the return of showCache is sorted by cacheId. For convenience, a user can optionally have it unsorted (passing sorted = FALSE), which may be noticeably faster when the cache is large (> 1e4 entries).

Value

Will clear all objects (or those that match userTags, or those between after or before) from the repository located in cachePath. Invisibly returns a data.table of the removed items.

Note

If the cache is larger than 10MB, and clearCache is used, there will be a message and a pause, if interactive, to prevent accidentally deleting of a large cache repository.

See Also

mergeCache(). Many more examples in Cache().

Examples

```
data.table::setDTthreads(2)

tmpDir <- file.path(tempdir(), "reproducible_examples", "Cache")
try(clearCache(tmpDir, ask = FALSE), silent = TRUE) # just to make sure it is clear

# Basic use
ranNumsA <- Cache(rnorm, 10, 16, cachePath = tmpDir)

# All same
ranNumsB <- Cache(rnorm, 10, 16, cachePath = tmpDir) # recovers cached copy
ranNumsD <- Cache(quote(rnorm(n = 10, 16)), cachePath = tmpDir) # recovers cached copy
```

```
# Any minor change makes it different
ranNumsE <- Cache(rnorm, 10, 6, cachePath = tmpDir) # different

## Example 1: basic cache use with tags
ranNumsA <- Cache(rnorm, 4, cachePath = tmpDir, userTags = "objectName:a")
ranNumsB <- Cache(runif, 4, cachePath = tmpDir, userTags = "objectName:b")
ranNumsC <- Cache(runif, 40, cachePath = tmpDir, userTags = "objectName:b")

showCache(tmpDir, userTags = c("objectName"))
showCache(tmpDir, userTags = c("^a$")) # regular expression ... "a" exactly

# Fine control of cache elements -- pick out only the large runif object, and remove it
cache1 <- showCache(tmpDir, userTags = c("runif")) # show only cached objects made during runif
toRemove <- cache1[tagKey == "object.size"][as.numeric(tagValue) > 700]$cacheId
clearCache(tmpDir, userTags = toRemove, ask = FALSE)
cacheAfter <- showCache(tmpDir, userTags = c("runif")) # Only the small one is left

data.table::setDTthreads(2)
tmpDir <- file.path(tempdir(), "reproducible_examples", "Cache")
try(clearCache(tmpDir, ask = FALSE), silent = TRUE) # just to make sure it is clear

Cache(rnorm, 1, cachePath = tmpDir)
thisTime <- Sys.time()
Cache(rnorm, 2, cachePath = tmpDir)
Cache(rnorm, 3, cachePath = tmpDir)
Cache(rnorm, 4, cachePath = tmpDir)
showCache(x = tmpDir) # shows all 4 entries
cc(ask = FALSE, x = tmpDir)
showCache(x = tmpDir) # most recent is gone
cc(thisTime, ask = FALSE, x = tmpDir)
showCache(x = tmpDir) # all those after thisTime gone, i.e., only 1 left
cc(ask = FALSE, x = tmpDir) # Cache is
cc(ask = FALSE, x = tmpDir) # Cache is already empty
```

`studyAreaName` *Get a unique name for a given study area*

Description

Digest a spatial object to get a unique character string (hash) of the study area. Use `.suffix()` to append the hash to a filename, e.g., when using `filename2` in `prepInputs`.

Usage

```
studyAreaName(studyArea, ...)

## S4 method for signature 'character'
studyAreaName(studyArea, ...)
```

```
## S4 method for signature 'ANY'
studyAreaName(studyArea, ...)
```

Arguments

studyArea	Spatial object.
...	Other arguments (not currently used)

Value

A character string using the .robustDigest of the studyArea. This is only intended for use with spatial objects.

Examples

```
studyAreaName("Ontario")
```

---

tempdir2 *Make a temporary (sub-)directory*

Description

Create a temporary subdirectory in getOption("reproducible.tempPath").

Usage

```
tempdir2(
  sub = "",
  tempdir = getOption("reproducible.tempPath", .reproducibleTempPath()),
  create = TRUE
)
```

Arguments

sub	Character string, length 1. Can be a result of file.path("smth", "smth2") for nested temporary subdirectories. If the zero length character, then a random sub-directory will be created.
tempdir	Optional character string where the temporary directory should be placed. Defaults to getOption("reproducible.tempPath").
create	Logical. Should the directory be created. Default TRUE.

Value

A character string of a path (that will be created if create = TRUE) in a sub-directory of the tempdir().

See Also

tempfile2

Examples

```
td <- tempdir2("myProject")
dir.exists(td)
unlink(td, recursive = TRUE)
```

`tempfile2` *Make a temporary file in a temporary (sub-)directory*

Description

Make a temporary file in a temporary (sub-)directory

Usage

```
tempfile2(
  sub = "",
  tempdir = getOption("reproducible.tempPath", .reproducibleTempPath()),
  ...
)
```

Arguments

`sub`	Character string, length 1. Can be a result of `file.path("smth", "smth2")` for nested temporary subdirectories. If the zero length character, then a random sub-directory will be created.
`tempdir`	Optional character string where the temporary directory should be placed. Defaults to `getOption("reproducible.tempPath")`.
`...`	passed to `tempfile`, e.g., `fileext`

Value

A character string of a path to a file in a sub-directory of the `tempdir()`. This file will likely not exist yet.

See Also

tempdir2

Examples

```
tf <- tempfile2("myProject", fileext = ".csv")
basename(tf)
unlink(dirname(tf), recursive = TRUE)
```

unrarPath *The known path for unrar or 7z*

Description

The known path for unrar or 7z

Usage

```
.systemArchivePath
```

usesPointer *Does an object use a pointer?*

Description

Does an object use a pointer?

Usage

```
usesPointer(x)
```

Arguments

x	an object

Value

logical

Examples

```
usesPointer(1:10) # FALSE for an ordinary vector
```

writeFuture *Write to cache repository, using* future::future *(defunct)*

Description

[Defunct]

Defunct as of version 3.2.0. The future-based cache writing this supported was never enabled by default – reproducible.futurePlan defaults to FALSE – and has been removed. Cache() writes directly.

Usage

```
writeFuture(
  written,
  outputToSave,
  cachePath,
  userTags,
  drv = getDrv(getOption("reproducible.drv", NULL)),
  conn = getOption("reproducible.conn", NULL),
  cacheId,
  linkToCacheId = NULL,
  verbose = getOption("reproducible.verbose")
)
```

Arguments

written	Integer. If zero or positive then it needs to be written still. Should be 0 to start.
outputToSave	The R object to save to repository
cachePath	The file path of the repository
userTags	Character string of tags to attach to this outputToSave in the CacheRepo
drv	If using a database backend, drv must be an object that inherits from DBIDriver (e.g., RSQLite::SQLite).
conn	an optional DBIConnection object, as returned by dbConnect().
cacheId	Character string. If passed, this will override the calculated hash of the inputs, and return the result from this cacheId in the cachePath. Setting this is equivalent to manually saving the output of this function, i.e., the object will be on disk, and will be recovered in subsequent This may help in some particularly finicky situations where Cache is not correctly detecting unchanged inputs. This will guarantee the object will be identical each time; this may be useful in operational code.
linkToCacheId	Optional. If a cacheId is provided here, then a file.link will be made to the file with that cacheId name in the cache repo. This is used when identical outputs exist in the cache. This will save disk space.

verbose — Numeric, -1 silent (where possible), 0 being very quiet, 1 showing more messaging, 2 being more messaging, etc. Default is 1. Above 3 will output much more information about the internals of Caching, which may help diagnose Caching challenges. Can set globally with an option, e.g., `options('reproducible.verbose' = 0)` to reduce t

Value

Nothing; it signals a defunct error.

Index

.addTagsRepo, 5
.addTagsRepo(), *7*
.debugCache, 7
.digest(), *39*
.file.move, 8
.isCRSany *(*.isGridded*)*, 8
.isGridded, 8
.isMemoised, 9
.isSF *(*.isGridded*)*, 8
.isSpat *(*.isGridded*)*, 8
.isSpatialAny *(*.isGridded*)*, 8
.isVector *(*.isGridded*)*, 8
.messageFunctionFn *(*messageDF*)*, 71
.objSizeWithTry, 10
.prefix, 10
.prepareFileBackedRaster, 11
.removeCacheAtts, 12
.requireNamespace, 13
.robustDigest(), *30*, *45*
.setSubAttrInList, 13
.suffix *(*.prefix*)*, 10
.systemArchivePath *(*unrarPath*)*, 126
.unwrap *(*.wrap*)*, 14
.updateTagsRepo *(*.addTagsRepo*)*, 5
.whereInStack, 14
.wrap, 14

asPath *(*Path-class*)*, 80
assessDataType, 18

base::basename(), *21*
base::search(), *118*
basename2, 21

Cache, 21
Cache(), *4*, *25*, *33*, *56*, *93*, *94*, *104*, *107*, *110–114*, *122*, *127*
cache2 *(*Cache*)*, 21
CacheDBFile *(*createCache*)*, 48
CacheDBTableName *(*createCache*)*, 48
CacheDigest, 32
CacheDigest(), *30*, *114*
CacheGeo, 33
cacheId, 36
CacheIsACache *(*createCache*)*, 48
CacheStorageDir *(*createCache*)*, 48
CacheStorageDir(), *30*
CacheStoredFile *(*createCache*)*, 48
CacheV2 *(*Cache*)*, 21
call, *70*
cc *(*showCache*)*, 119
checkPath, 37
checkPath,character,logical-method *(*checkPath*)*, 37
checkPath,character,missing-method *(*checkPath*)*, 37
checkPath,missing,ANY-method *(*checkPath*)*, 37
checkPath,NULL,ANY-method *(*checkPath*)*, 37
checkRelative, 38
Checksums, 39
Checksums(), *93*, *104*
Checksums,character,logical-method *(*Checksums*)*, 39
Checksums,character,missing-method *(*Checksums*)*, 39
clearCache *(*showCache*)*, 119
clearCache(), *26*, *30*, *107*
clearCache,ANY-method *(*showCache*)*, 119
clearUrlLog *(*prepInputsLog*)*, 98
clearUrlLog(), *112*
cloudDownload, 41
coerce,character,Path-method *(*Path-class*)*, 80
compareNA, 42
convertCallToCommonFormat, 43
convertPaths, 43
convertRasterPaths *(*convertPaths*)*, 43

Copy, 44
Copy,ANY-method *(Copy)*, 44
Copy,data.frame-method *(Copy)*, 44
Copy,data.table-method *(Copy)*, 44
Copy,list-method *(Copy)*, 44
Copy,refClass-method *(Copy)*, 44
copyFile *(copySingleFile)*, 46
copySingleFile, 46
createCache, 48
cropInputs *(postProcessInputsSuperseded)*, 85
cropInputs(), *56*, *87*, *94*, *104*
cropTo *(postProcessTo)*, 87
cropTo(), *82*, *85*, *90, 91*, *94*

dataType2 *(minFn)*, 73
detectActiveCores, 52
detectActiveCores(), *78*
determineFilename, 52
determineFilename(), *53*, *82*
digest::digest(), *24*, *26*, *33*, *40*
dir.create(), *38*
downloadFile(), *96*, *110*
downloadRemote, 54

extractFromArchive, 56
extractFromArchive(), *96*
extractFromCache *(createCache)*, 48

file.copy(), *65*
file.exists(), *38*
file.link(), *65*
file.symlink(), *65*
Filenames, 57
Filenames(), *45*
Filenames,ANY-method *(Filenames)*, 57
Filenames,data.table-method *(Filenames)*, 57
Filenames,environment-method *(Filenames)*, 57
Filenames,list-method *(Filenames)*, 57
Filenames,Path-method *(Filenames)*, 57
fixErrorsIn, 58
fixErrorsIn(), *82*, *91*

gdalMask *(gdalProject)*, 59
gdalMask(), *60*, *91*
gdalProject, 59
gdalProject(), *91*
gdalResample *(gdalProject)*, 59
gdalResample(), *60*, *91*
getRelative, 61
googledrive::drive_get(), *101*

harmonizeCall, 62

isSpatial *(.isGridded)*, 8
isUpdated, 63

keepCache *(showCache)*, 119
keepCache(), *30*, *107*
keepCache,ANY-method *(showCache)*, 119
keepOrigGeom, 64

linkOrCopy, 64
listGoogleDriveFolder, 66
listNamed, 67
loadFile(), *51*
loadFromCache *(createCache)*, 48

makeRelative *(getRelative)*, 61
makeUrlRemap, 68
makeUrlRemap(), *66, 67*, *112*
maskInputs *(postProcessInputsSuperseded)*, 85
maskTo *(postProcessTo)*, 87
maskTo(), *60*, *82*, *85*, *87*, *90, 91*, *94*
matchCall2, 69
maxFn *(minFn)*, 73
mergeCache, 70
mergeCache(), *122*
mergeCache,ANY-method *(mergeCache)*, 70
messageCache *(messageDF)*, 71
messageColoured *(messageDF)*, 71
messageDF, 71
messagePrepInputs *(messageDF)*, 71
messagePreProcess *(messageDF)*, 71
messagePreProcess(), *108*
messageQuestion *(messageDF)*, 71
minFn, 73
movedCache, 74
movedCache(), *30*

nlayers2 *(minFn)*, 73
normPath, 75
normPath(), *38*
normPath,character-method *(normPath)*, 75

normPath,list-method *(normPath)*, 75
normPath,logical-method *(normPath)*, 75
normPath,missing-method *(normPath)*, 75
normPath,NULL-method *(normPath)*, 75
normPathRel *(normPath)*, 75
numCoresToUse, 77

objSize, 78
objSizeSession *(objSize)*, 78
options(), *5*

paddedFloatToChar, 79
Path-class, 80
postProcess, 82
postProcess(), *56*, *87*, *90, 91*, *94–96*, *104*, *109, 110*
postProcessInputsSuperseded, 85
postProcessTerra *(postProcessTo)*, 87
postProcessTo, 87
postProcessTo(), *60*, *64*, *83*, *85–87*, *94–96*, *109*
prepInputs, 92
prepInputs(), *4*, *53*, *66, 67*, *108–110*
prepInputsLog, 98
prepInputsLog(), *112*
prepInputsWithTiles, 99
prepopulateCacheAsync, 101
preProcess *(preProcessParams)*, 102
preProcess(), *54–56*, *68, 69*, *92–94*, *103*, *108–110*
preProcessParams, 102
projectInputs *(postProcessInputsSuperseded)*, 85
projectTo *(postProcessTo)*, 87
projectTo(), *82*, *85–87*, *90, 91*, *94*
purgeChecksums, 105

rasterRead, 106
remapFilenames, 106
reproducible *(reproducible-package)*, 4
reproducible-package, 4
reproducibleOptions, *24*, 107
reproducibleOptions(), *5*, *26*, *30*, *69*
retry, 115
rmFromCache *(createCache)*, 48
rmFromCache(), *30*

saveToCache, 116
searchFull, 117
searchFullEx *(searchFull)*, 117
set.randomseed, 118
showCache, 119
showCache(), *25*, *30*, *107*, *117*
showCache,ANY-method *(showCache)*, 119
studyAreaName, 123
studyAreaName,ANY-method *(studyAreaName)*, 123
studyAreaName,character-method *(studyAreaName)*, 123
suffix *(.prefix)*, 10

tempdir2, 124, *125*
tempfile2, *124*, 125
terra::crop(), *101*
terra::merge(), *101*
terra::rast(), *101*
terra::terraOptions(), *110*

unrarPath, 126
untar(), *94*
unzip(), *94*
usesPointer, 126
utils::download.file(), *94*
utils::write.table(), *40*

values2 *(minFn)*, 73

writeFuture, 127
writeTo *(postProcessTo)*, 87
writeTo(), *82, 83*, *90, 91*, *94*