Package 'drake'

January 26, 2018

Title Data Frames in R for Make

Description An R-focused pipeline toolkit
for reproducible code and high-performance computing.

Version 5.0.0

License GPL-3

URL https://github.com/ropensci/drake

BugReports https://github.com/ropensci/drake/issues

Depends R (>= 3.2.0)

Imports codetools, crayon, evaluate, digest, formatR, future,
future.apply, grDevices, igraph, knitr, lubridate, magrittr,
parallel, plyr, R.utils, rprojroot, stats, storr (>= 1.1.0),
stringi, stringr, testthat, utils, visNetwork, withr

Suggests abind, cranlogs, DBI, Ecdat, future.batchtools, ggplot2,
MASS, methods, RSQLite, rmarkdown, tibble

VignetteBuilder knitr

RoxygenNote 6.0.1

NeedsCompilation no

Author William Michael Landau [aut, cre],
Alex Axthelm [ctb],
Jasper Clarkberg [ctb],
Ben Marwick [rev],
Peter Slaughter [rev],
Eli Lilly and Company [cph]

Maintainer William Michael Landau <will.landau@gmail.com>

Repository CRAN

Date/Publication 2018-01-26 11:22:52 UTC

R topics documented:

---

drake-package *Drake is a pipeline toolkit (https://github.com/pditommaso/awesome-pipeline) and a scalable, R-focused solution for reproducibility and high-performance computing.*

Description

Drake is a pipeline toolkit (https://github.com/pditommaso/awesome-pipeline) and a scalable, R-focused solution for reproducibility and high-performance computing.

Author(s)

William Michael Landau <will.landau@lilly.com>

References

https://github.com/ropensci/drake

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
library(drake)
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Build everything.
make(my_plan) # Nothing is done because everything is already up to date.
reg2 = function(d){ # Change one of your functions.
  d$x3 = d$x^3
  lm(y ~ x3, data = d)
}
make(my_plan) # Only the pieces depending on reg2() get rebuilt.
# Write a flat text log file this time.
make(my_plan, cache_log_file = TRUE)
# Read/load from the cache.
readd(small)
loadd(large)
head(large)
clean() # Restart from scratch.
make(my_plan, jobs = 2) # Distribute over 2 parallel jobs.
clean() # Restart from scratch.
# Parallelize over at most 4 separate R sessions.
# Requires Rtools on Windows.
# make(my_plan, jobs = 4, parallelism = "Makefile") # nolint
# Everything is already up to date.
# make(my_plan, jobs = 4, parallelism = "Makefile") # nolint
```

```
clean(destroy = TRUE) # Totally remove the cache.
unlink("report.Rmd") # Clean up the remaining files.
})

## End(Not run)
```

`analysis_wildcard` *Show the analysis wildcard used in* `plan_summaries()`.

Description

Used to generate workflow plan data frames.

Usage

```
analysis_wildcard()
```

Value

The analysis wildcard used in `plan_summaries()`.

See Also

`plan_summaries()`

Examples

```
# See ?plan_analyses for examples
```

`as_drake_filename` *Converts an ordinary character string into a filename understandable by drake.*

Description

This function simply wraps single quotes around `x`. Quotes are important in drake. In workflow plan data frame commands, single-quoted targets denote physical files, and double-quoted strings are treated as ordinary string literals.

Usage

```
as_drake_filename(x)
```

Arguments

`x`	character string to be turned into a filename understandable by drake (i.e., a string with literal single quotes on both ends).

Value

A single-quoted character string: i.e., a filename understandable by drake.

Examples

```
# Wraps the string in single quotes.
as_drake_filename("my_file.rds") # "'my_file.rds'"
```

`available_hash_algos` *List the available hash algorithms for drake caches.*

Description

See the advanced storage tutorial at https://ropensci.github.io/drake/articles/storage.html for details.

Usage

```
available_hash_algos()
```

Value

A character vector of names of available hash algorithms.

Examples

```
available_hash_algos()
```

`build_drake_graph` *Create the* `igraph` *dependency network of your project.*

Description

This function returns an igraph object representing how the targets in your workflow plan data frame depend on each other. (`help(package = "igraph")`). To plot the graph, call to `plot.igraph()` on your graph, or just use `vis_drake_graph()` from the start.

Usage

```
build_drake_graph(plan = drake_plan(),
  targets = drake::possible_targets(plan), envir = parent.frame(),
  verbose = 1, jobs = 1)
```

Arguments

plan	workflow plan data frame, same as for function make().
targets	names of targets to build, same as for function make().
envir	environment to import from, same as for function make().
verbose	logical, whether to output messages to the console.
jobs	number of jobs to accelerate the construction of the dependency graph. A light mclapply-based parallelism is used if your operating system is not Windows.

Value

An igraph object representing the workflow plan dependency network.

See Also

vis_drake_graph

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Make the igraph network connecting all the targets and imports.
g <- build_drake_graph(my_plan)
class(g) # "igraph"
})

## End(Not run)
```

---

build_times *List the time it took to build each target/import.*

Description

Listed times do not include the amount of time spent loading and saving objects!

Usage

```
build_times(path = getwd(), search = TRUE, digits = 3,
  cache = get_cache(path = path, search = search, verbose = verbose),
  targets_only = FALSE, verbose = TRUE, jobs = 1)
```

Arguments

path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
digits	How many digits to round the times to.
cache	optional drake cache. If supplied, the path and search arguments are ignored.
targets_only	logical, whether to only return the build times of the targets (exclude the imports).
verbose	whether to print console messages
jobs	number of parallel jobs/workers for light parallelism.

Value

A data frame of times, each from system.time().

See Also

built

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# Show the build times for the basic example.
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Build all the targets.
build_times() # Show how long it took to build each target.
})

## End(Not run)
```

---

built *List all the built targets (non-imports) in the cache.*

Description

Targets are listed in the workflow plan data frame (see drake_plan().

Usage

```
built(path = getwd(), search = TRUE, cache = drake::get_cache(path = path,
  search = search, verbose = verbose), verbose = TRUE, jobs = 1)
```

Arguments

path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	drake cache. See new_cache(). If supplied, path and search are ignored.
verbose	whether to print console messages
jobs	number of jobs/workers for parallel processing

Value

Character vector naming the built targets in the cache.

See Also

cached, loadd, link{imported}

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Load drake's canonical example.
make(my_plan) # Run the project, build all the targets.
built() # List all the cached targets (built objects and files).
# For file targets, only the fingerprints/hashes are stored.
})

## End(Not run)
```

---

cached *Enumerate cached targets or check if a target is cached.*

Description

Read/load a cached item with readd() or loadd().

Usage

```
cached(..., list = character(0), no_imported_objects = FALSE,
  path = getwd(), search = TRUE, cache = NULL, verbose = 1,
  namespace = NULL, jobs = 1)
```

Arguments

...	objects to load from the cache, as names (unquoted) or character strings (quoted). Similar to ... in remove(...).
list	character vector naming objects to be loaded from the cache. Similar to the list argument of remove().
no_imported_objects	logical, applies only when no targets are specified and a list of cached targets is returned. If no_imported_objects is TRUE, then cached() shows built targets (with commands) plus imported files, ignoring imported objects. Otherwise, the full collection of all cached objects will be listed. Since all your functions and all their global variables are imported, the full list of imported objects could get really cumbersome.
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project. Ignored if a cache is supplied.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a cache is supplied.
cache	drake cache. See new_cache(). If supplied, path and search are ignored.
verbose	whether to print console messages
namespace	character scalar, name of the storr namespace to use for listing objects
jobs	number of jobs/workers for parallel processing

Value

Either a named logical indicating whether the given targets or cached or a character vector listing all cached items, depending on whether any targets are specified

See Also

built, imported, readd, loadd, drake_plan, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Load drake's canonical example.
make(my_plan) # Run the project, build all the targets.
cached(list = 'reg1') # Is 'reg1' in the cache?
# List all the targets and imported files in the cache.
# Exclude R objects imported from your workspace.
cached(no_imported_objects = TRUE)
# List all targets and imports in the cache.
cached()
# Clean the main data.
clean()
# The targets and imports are gone.
cached()
# But there is still metadata.
build_times()
```

```
cached(namespace = "build_times")
# Clear that too.
clean(purge = TRUE)
cached(namespace = "build_times")
build_times()
})

## End(Not run)
```

---

cache_namespaces *List all the* storr *cache namespaces used by drake.*

Description

Ordinary users do not need to worry about this function. It is just another window into drake's internals.

Usage

```
cache_namespaces(default = storr::storr_environment()$default_namespace)
```

Arguments

default	name of the default storr namespace

Value

A character vector of storr namespaces used for drake.

See Also

make

Examples

```
cache_namespaces()
```

cache_path *Return the file path where the cache is stored, if applicable.*

Description

Currently only works with `storr::storr_rds` file system caches.

Usage

```
cache_path(cache = NULL)
```

Arguments

cache	the cache whose file path you want to know

Value

File path where the cache is stored.

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
clean(destroy = TRUE)
# Get/create a new drake/storr cache.
cache <- recover_cache()
# Show the file path of the cache.
cache_path(cache = cache)
# In-memory caches do not have file paths.
mem <- storr_environment()
cache_path(cache = mem)
})

## End(Not run)
```

check_plan *Check a workflow plan data frame for obvious errors.*

Description

Possible obvious errors include circular dependencies and missing input files.

Usage

```
check_plan(plan = drake_plan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), cache = drake::get_cache(verbose = verbose),
  verbose = TRUE, jobs = 1)
```

Arguments

plan	workflow plan data frame, possibly from drake_plan().
targets	character vector of targets to make
envir	environment containing user-defined functions
cache	optional drake cache. See new_cache()
verbose	same as for make()
jobs	number of jobs/workers for light parallelism

Value

Invisibly return plan.

See Also

ink{drake_plan}, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
check_plan(my_plan) # Check the workflow plan dataframe for obvious errors.
unlink('report.Rmd') # Remove an import file mentioned in the plan.
# If you un-suppress the warnings, check_plan()
# will tell you that 'report.Rmd' is missing.
suppressWarnings(check_plan(my_plan))
})

## End(Not run)
```

---

clean *Remove targets/imports from the cache.*

Description

Cleans up the work done by make().

Usage

```
clean(..., list = character(0), destroy = FALSE, path = getwd(),
  search = TRUE, cache = NULL, verbose = 1, jobs = 1, force = FALSE,
  garbage_collection = FALSE, purge = FALSE)
```

Arguments

...	targets to remove from the cache, as names (unquoted) or character strings (quoted). Similar to ... in remove(...). The symbols must not match other (formal) arguments of clean(), such as destroy, cache, path, search, verbose, or jobs. If there are name conflicts, use the list argument instead of ....
list	character vector naming targets to be removed from the cache. Similar to the list argument of remove().
destroy	logical, whether to totally remove the drake cache. If destroy is FALSE, only the targets from make() are removed. If TRUE, the whole cache is removed, including session metadata, etc.
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	optional drake cache. See codenew_cache(). If If cache is supplied, the path and search arguments are ignored.
verbose	whether to print console messages
jobs	Number of jobs for light parallelism (disabled on Windows).
force	logical, whether to try to clean the cache even though the project may not be back compatible with the current version of drake.
garbage_collection	logical, whether to call cache$gc() to do garbage collection. If TRUE, cached data with no remaining references will be removed. This will slow down clean(), but the cache could take up far less space afterwards. See the gc() method for storr caches.
purge	logical, whether to remove objects from metadata namespaces such as "meta", "build_times", and "errors".

Details

By default, clean() removes references to cached data. To deep-clean the data to free up storage/memory, use clean(garbage_collection = TRUE). Garbage collection is slower, but it purges data with no remaining references. To just do garbage collection without cleaning, see drake_gc(). Also, for clean(), you must be in your project's working directory or a subdirectory of it. clean(search = TRUE) searches upwards in your folder structure for the drake cache and acts on the first one it sees. Use search == FALSE to look within the current working directory only. WARNING: This deletes ALL work done with make(), which includes file targets as well as the entire drake cache. Only use clean() if you're sure you won't lose anything important.

Value

Invisibly return NULL.

See Also

drake_gc, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
# List objects in the cache, excluding R objects
# imported from your workspace.
cached(no_imported_objects = TRUE)
# Remove 'summ_regression1_large' and 'small' from the cache.
clean(summ_regression1_large, small)
# Those objects should be gone.
cached(no_imported_objects = TRUE)
# Rebuild the missing targets.
make(my_plan)
# Remove all the targets and imports.
# On non-Windows machines, parallelize over at most 2 jobs.
clean(jobs = 2)
# Make the targets again.
make(my_plan)
# Garbage collection removes data whose references are no longer present.
# It is slow, but you should enable it if you want to reduce the
# size of the cache.
clean(garbage_collection = TRUE)
# All the targets and imports are gone.
cached()
# But there is still cached metadata.
names(read_drake_meta())
build_times()
# To make even more room, use the "purge" flag.
clean(purge = TRUE)
names(read_drake_meta())
build_times()
# Completely remove the entire cache (default: '.drake/' folder).
clean(destroy = TRUE)
})

## End(Not run)
```

`cleaned_namespaces` *For drake caches, list the* `storr` *namespaces that are cleaned during a call to* `clean()`.

Description

All these namespaces store target-level data, but not all target-level namespaces are cleaned during `clean()`.

Usage

```
cleaned_namespaces(default = storr::storr_environment()$default_namespace)
```

Arguments

default	Name of the default storr namespace.

Value

A character vector of storr namespaces that are cleaned during clean().

See Also

cache_namespaces, clean

Examples

```
cleaned_namespaces()
```

---

configure_cache *Configure the hash algorithms, etc. of a drake cache.*

Description

The purpose of this function is to prepare the cache to be called from make().

Usage

```
configure_cache(cache = drake::get_cache(verbose = verbose),
  short_hash_algo = drake::default_short_hash_algo(cache = cache),
  long_hash_algo = drake::default_long_hash_algo(cache = cache),
  log_progress = FALSE, overwrite_hash_algos = FALSE, verbose = TRUE,
  jobs = 1)
```

Arguments

cache	cache to configure
short_hash_algo	short hash algorithm for drake. The short algorithm must be among available_hash_algos{}, which is just the collection of algorithms available to the 'algo' argument in digest::digest(). See ?default_short_hash_algo for more.
long_hash_algo	short hash algorithm for drake. The long algorithm must be among available_hash_algos{}, which is just the collection of algorithms available to the 'algo' argument in digest::digest(). See ?default_long_hash_algo for more.
log_progress	logical, whether to clear the recorded build progress if this cache was used for previous calls to make()
overwrite_hash_algos	logical, whether to try to overwrite the hash algorithms in the cache with any user-specified ones.
verbose	whether to print console messages
jobs	number of jobs for parallel processing

Value

A drake/storr cache.

See Also

default_short_hash_algo, default_long_hash_algo

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
clean(destroy = TRUE)
load_basic_example() # Get the code with drake_example("basic").
config <- make(my_plan) # Run the project, build all the targets.
# Locate the drake/storr cache of the project
# inside the master internal configuration list.
cache <- config$cache
long_hash(cache) # Return the long hash algorithm used.
# Change the long hash algorithm of the cache.
cache <- configure_cache(
  cache = cache,
  long_hash_algo = "murmur32",
  overwrite_hash_algos = TRUE
)
long_hash(cache) # Show the new long hash algorithm.
make(my_plan) # Changing the long hash puts the targets out of date.
})

## End(Not run)
```

---

dataframes_graph *Create the underlying node and edge data frames behind* vis_drake_graph().

Description

With the returned data frames, you can plot your own custom visNetwork graph.

Usage

```
dataframes_graph(config, from = NULL, mode = c("out", "in", "all"),
  order = NULL, subset = NULL, build_times = TRUE, digits = 3,
  targets_only = FALSE, split_columns = FALSE, font_size = 20,
  from_scratch = FALSE, make_imports = TRUE)
```

Arguments

config	a drake_config() configuration list. You can get one as a return value from make() as well.
from	Optional collection of target/import names. If from is nonempty, the graph will restrict itself to a neighborhood of from. Control the neighborhood with mode and order.
mode	Which direction to branch out in the graph to create a neighborhood around from. Use "in" to go upstream, "out" to go downstream, and "all" to go both ways and disregard edge direction altogether.
order	How far to branch out to create a neighborhood around from (measured in the number of nodes). Defaults to as far as possible.
subset	Optional character vector of of target/import names. Subset of nodes to display in the graph. Applied after from, mode, and order. Be advised: edges are only kept for adjacent nodes in subset. If you do not select all the intermediate nodes, edges will drop from the graph.=
build_times	logical, whether to show the build_times() of the targets and imports, if available. These are just elapsed times from system.time().
digits	number of digits for rounding the build times
targets_only	logical, whether to skip the imports and only include the targets in the workflow plan.
split_columns	logical, whether to break up the columns of nodes to make the aspect ratio of the rendered graph closer to 1:1. This improves the viewing experience, but the columns no longer strictly represent parallelizable stages of build items. (Although the targets/imports in each column are still conditionally independent, there may be more conditional independence than the graph indicates.)
font_size	numeric, font size of the node labels in the graph
from_scratch	logical, whether to assume all the targets will be made from scratch on the next make(). Makes all targets outdated, but keeps information about build progress in previous make()s.
make_imports	logical, whether to make the imports first. Set to FALSE to increase speed and risk using obsolete information.

Value

A list of three data frames: one for nodes, one for edges, and one for the legend nodes. The list also contains the default title of the graph.

See Also

vis_drake_graph, build_drake_graph

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
```

```
config <- load_basic_example() # Get the code with drake_example("basic").
# Get a list of data frames representing the nodes, edges,
# and legend nodes of the visNetwork graph from vis_drake_graph().
raw_graph <- dataframes_graph(config = config)
# Choose a subset of the graph.
smaller_raw_graph <- dataframes_graph(
  config = config,
  from = c("small", "reg2"),
  mode = "in"
)
# Inspect the raw graph.
str(raw_graph)
# Use the data frames to plot your own custom visNetwork graph.
# For example, you can omit the legend nodes
# and change the direction of the graph.
library(magrittr)
library(visNetwork)
visNetwork(nodes = raw_graph$nodes, edges = raw_graph$edges) %>%
  visHierarchicalLayout(direction = 'UD')
})

## End(Not run)
```

`dataset_wildcard` *Show the dataset wildcard used in* `plan_analyses()` *and* `plan_summaries()`.

Description

Used to generate workflow plan data frames.

Usage

```
dataset_wildcard()
```

Value

The dataset wildcard used in `plan_analyses()` and `plan_summaries()`.

See Also

`plan_analyses()`

Examples

```
# See ?plan_analyses for examples
```

default_cache_path *Return the default file path of the drake/storr cache.*

Description

Applies to file system caches only.

Usage

```
default_cache_path()
```

Value

Default file path of the drake/storr cache.

Examples

```
default_cache_path()
```

default_hook *Default* hook *argument to* make().

Description

Most users do not need to micromanage hooks.

Usage

```
default_hook(code)
```

Arguments

code	Placeholder for the code to build a target/import.

Value

A function that you can supply to the hook argument of make().

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Everything gets built normally.
make(my_plan, hook = default_hook)
cached() # List the cached targets and imports.
})

## End(Not run)
```

default_long_hash_algo *Return the default long hash algorithm for* make().

Description

See the advanced storage tutorial at https://ropensci.github.io/drake/articles/storage.html for details.

Usage

```
default_long_hash_algo(cache = NULL)
```

Arguments

cache	optional drake cache. When you configure_cache(cache) without supplying a long hash algorithm, default_long_hash_algo(cache) is the long hash algorithm that drake picks for you.

Details

The long algorithm must be among available_hash_algos{}, which is just the collection of algorithms available to the 'algo' argument in digest::digest().

If you express no preference for a hash, drake will use the long hash for the existing project, or default_long_hash_algo() for a new project. If you do supply a hash algorithm, it will only apply to fresh projects (see clean(destroy = TRUE)). For a project that already exists, if you supply a hash algorithm, drake will warn you and then ignore your choice, opting instead for the hash algorithm already chosen for the project in a previous make().

Drake uses both a short hash algorithm and a long hash algorithm. The shorter hash has fewer characters, and it is used to generate the names of internal cache files and auxiliary files. The decision for short names is important because Windows places restrictions on the length of file paths. On the other hand, some internal hashes in drake are never used as file names, and those hashes can use a longer hash to avoid collisions.

Value

A character vector naming a hash algorithm.

See Also

make, available_hash_algos

Examples

```
default_long_hash_algo()
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Run the project and return the internal master configuration list.
config <- make(my_plan)
# Locate the storr cache.
cache <- config$cache
# Get the default long hash algorithm of an existing cache.
default_long_hash_algo(cache)
})

## End(Not run)
```

`default_Makefile_args` *Return the default value of the* `args` *argument to* `make()`.

Description

For `make(..., parallelism = "Makefile")`, this function configures the default arguments to `system2()`. It is an internal function, and most users do not need to worry about it.

Usage

```
default_Makefile_args(jobs, verbose)
```

Arguments

`jobs`	number of jobs
`verbose`	logical, whether to be verbose

Value

`args` for `system2(command, args)`

Examples

```
default_Makefile_args(jobs = 2, verbose = FALSE)
default_Makefile_args(jobs = 4, verbose = TRUE)
```

default_Makefile_command *Give the default* command *argument to* make().

Description

Relevant for "Makefile" parallelism only.

Usage

```
default_Makefile_command()
```

Value

A character scalar naming a Linux/Unix command to run a Makefile.

Examples

```
default_Makefile_command()
```

default_parallelism *Show the default* parallelism *argument to* make() *for your system.*

Description

Returns 'parLapply' for Windows machines and 'mclapply' for other platforms.

Usage

```
default_parallelism()
```

Value

The default parallelism option for your system.

See Also

make, shell_file

Examples

```
default_parallelism()
```

`default_recipe_command` *Show the default recipe command for* `make(..., parallelism = "Makefile")`.

Description

See the help file of `Makefile_recipe` for details and examples.

Usage

```
default_recipe_command()
```

Value

A character scalar with the default recipe command.

See Also

`Makefile_recipe`

Examples

```
default_recipe_command()
```

`default_short_hash_algo` *Return the default short hash algorithm for* `make()`.

Description

See the advanced storage tutorial at https://ropensci.github.io/drake/articles/storage.html for details.

Usage

```
default_short_hash_algo(cache = NULL)
```

Arguments

cache	optional drake cache. When you `configure_cache(cache)` without supplying a short hash algorithm, `default_short_hash_algo(cache)` is the short hash algorithm that drake picks for you.

Details

The short algorithm must be among `available_hash_algos{}`, which is just the collection of algorithms available to the 'algo' argument in `digest::digest()`.

If you express no preference for a hash, drake will use the short hash for the existing project, or `default_short_hash_algo()` for a new project. If you do supply a hash algorithm, it will only apply to fresh projects (see `clean(destroy = TRUE)`). For a project that already exists, if you supply a hash algorithm, drake will warn you and then ignore your choice, opting instead for the hash algorithm already chosen for the project in a previous `make()`.

Drake uses both a short hash algorithm and a long hash algorithm. The shorter hash has fewer characters, and it is used to generate the names of internal cache files and auxiliary files. The decision for short names is important because Windows places restrictions on the length of file paths. On the other hand, some internal hashes in drake are never used as file names, and those hashes can use a longer hash to avoid collisions.

Value

A character vector naming a hash algorithm.

See Also

`make`, `available_hash_algos`

Examples

```
default_short_hash_algo()
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Run the project and return the internal master configuration list.
config <- make(my_plan)
# Locate the storr cache.
cache <- config$cache
# Get the default short hash algorithm of an existing cache.
default_short_hash_algo(cache)
})

## End(Not run)
```

`default_trigger` *Return the default trigger.*

Description

Triggers are target-level rules that tell `make()` how to check if target is up to date or outdated.

Usage

```
default_trigger()
```

Value

A character scalar naming the default trigger.

See Also

triggers, make

Examples

```
default_trigger()
# See ?triggers for more examples.
```

---

dependency_profile *Return the detailed dependency profile of the target.*

Description

Useful for debugging. For up to date targets, like elements of the returned list should agree: for example, cached_dependency_hash and current_dependency_hash.

Usage

```
dependency_profile(target, config)
```

Arguments

target	name of the target
config	configuration list output by config or make

Value

A list of information that drake takes into account when examining the dependencies of the target.

See Also

read_drake_meta, deps, make, config

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Load drake's canonical exmaple.
con <- make(my_plan) # Run the project, build the targets.
# Get some example dependency profiles of targets.
dependency_profile("small", config = con)
dependency_profile("'report.md'", config = con)
})

## End(Not run)
```

`deps` *List the dependencies of a function, workflow plan command, or knitr report source file.*

Description

Intended for debugging and checking your project. The dependency structure of the components of your analysis decides which targets are built and when.

Usage

```
deps(x)
```

Arguments

`x`	Either a function or a string. Strings are commands from your workflow plan data frame.

Details

If the argument is a single-quoted string that points to a dynamic knitr report, the dependencies of the expected compiled output will be given. For example, `deps("'report.Rmd'")` will return target names found in calls to `loadd()` and `readd()` in active code chunks. These targets are needed in order to run `knit('report.Rmd')` to produce the output file `'report.md'`, so technically, they are dependencies of `'report.md'`, not `'report.Rmd'`

Value

A character vector, names of dependencies. Files wrapped in single quotes. The other names listed are functions or generic R objects.

Examples

```
# Your workflow likely depends on functions in your workspace.
f <- function(x, y){
  out <- x + y + g(x)
  saveRDS(out, 'out.rds')
}
# Find the dependencies of f. These could be R objects/functions
# in your workspace or packages. Any file names or target names
# will be ignored.
deps(f)
# Define a workflow plan data frame that uses your function f().
my_plan <- drake_plan(
  x = 1 + some_object,
  my_target = x + readRDS('tracked_input_file.rds'),
  return_value = f(x, y, g(z + w))
)
# Get the dependencies of workflow plan commands.
# Here, the dependencies could be R functions/objects from your workspace
# or packages, imported files, or other targets in the workflow plan.
deps(my_plan$command[1])
deps(my_plan$command[2])
deps(my_plan$command[3])
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Dependencies of the knitr-generated targets like 'report.md'
# include targets/imports referenced with `readd()` or `loadd()`.
deps("'report.Rmd'")
})

## End(Not run)
```

`diagnose` *Get the last stored error log of a target that failed to build, or list the targets with error logs.*

Description

The specified target could be a completely failed target or a target that failed initially, retried, then succeeded. If no target is given, then `diagnose()` simply lists the targets for which a error is retrievable. Together, functions `failed()` and `diagnose()` should eliminate the strict need for ordinary error messages printed to the console.

Usage

```
diagnose(target = NULL, character_only = FALSE, path = getwd(),
  search = TRUE, cache = drake::get_cache(path = path, search = search,
  verbose = verbose), verbose = TRUE)
```

Arguments

target	name of the target of the error to get. Can be a symbol if character_only is FALSE, must be a character if character_only is TRUE.
character_only	logical, whether target should be treated as a character or a symbol. Just like character.only in library().
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.
verbose	whether to print console messages

Value

Either a character vector of target names or an object of class "error".

See Also

failed, progress, readd, drake_plan, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
diagnose() # List all the targets with recorded error logs.
# Define a function doomed to failure.
f <- function(){
  stop("unusual error")
}
# Create a workflow plan doomed to failure.
bad_plan <- drake_plan(my_target = f())
# Running the project should generate an error
# when trying to build 'my_target'.
try(make(bad_plan), silent = FALSE)
failed() # List the failed targets from the last make() (my_target).
# List targets that failed at one point or another
# over the course of the project (my_target).
# Drake keeps all the error logs.
diagnose()
# Get the error log, an object of class "error".
error <- diagnose(my_target)
str(error) # See what's inside the error log.
error$calls # View the traceback. (See the traceback() function).
})

## End(Not run)
```

drake_batchtools_tmpl_file *Write the batchtools template file from one of the built-in drake examples.*

Description

If there are multiple template files in the example, only the first one (alphabetically) is written.

Usage

```
drake_batchtools_tmpl_file(example = drake::drake_examples(), to = getwd(),
  overwrite = FALSE)
```

Arguments

example	Name of the drake example from which to take the template file. Must be listed in drake_examples().
to	Character vector, where to write the file.
overwrite	Logical, whether to overwrite an existing file of the same name.

Value

NULL is returned, but a batchtools template file is written.

See Also

drake_examples, drake_example, shell_file

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# List the drake examples. Only some have template files.
drake_examples()
# Write the batchtools template file from the SLURM example.
drake_batchtools_tmpl_file("slurm") # Writes batchtools.slurm.tmpl.
# Find batchtools.slurm.tmpl with the rest of the example's files.
drake_example("slurm") # Writes a new 'slurm' folder with more files.
# Run the basic example with a
# SLURM-powered parallel backend. Requires SLURM.
library(future.batchtools)
# future::plan(batchtools_slurm, template = "batchtools.slurm.tmpl") # nolint
# make(my_plan, parallelism = "future_lapply") # nolint
})

## End(Not run)
```

drake_cache_log *Get a table that represents the state of the cache.*

Description

This functionality is like `make(..., cache_log_file = TRUE)`, but separated and more customizable. Hopefully, this functionality is a step toward better data versioning tools.

Usage

```
drake_cache_log(path = getwd(), search = TRUE,
  cache = drake::get_cache(path = path, search = search, verbose = verbose),
  verbose = TRUE, jobs = 1, targets_only = FALSE)
```

Arguments

`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project.
`search`	logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
`cache`	drake cache. See `new_cache()`. If supplied, `path` and `search` are ignored.
`verbose`	whether to print console messages
`jobs`	number of jobs/workers for parallel processing
`targets_only`	logical, whether to output information only on the targets in your workflow plan data frame. If `targets_only` is `FALSE`, the output will include the hashes of both targets and imports.

Details

A hash is a fingerprint of an object's value. Together, the hash keys of all your targets and imports represent the state of your project. Use `drake_cache_log()` to generate a data frame with the hash keys of all the targets and imports stored in your cache. This function is particularly useful if you are storing your drake project in a version control repository. The cache has a lot of tiny files, so you should not put it under version control. Instead, save the output of `drake_cache_log()` as a text file after each `make()`, and put the text file under version control. That way, you have a changelog of your project's results. See the examples below for details. Depending on your project's history, the targets may be different than the ones in your workflow plan data frame. Also, the keys depend on the short hash algorithm of your cache (default: `default_short_hash_algo()`).

Value

Data frame of the hash keys of the targets and imports in the cache

See Also

`drake_cache_log_file` `cached`, `get_cache`, `default_short_hash_algo`, `default_long_hash_algo`, `short_hash`, `long_hash`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# Load drake's canonical example.
load_basic_example() # Get the code with drake_example()
# Run the project, build all the targets.
make(my_plan)
# Get a data frame of all the hash keys.
# If you want a changelog, be sure to do this after every make().
cache_log <- drake_cache_log()
head(cache_log)
# Save the hash log as a flat text file.
write.table(
  x = cache_log,
  file = "drake_cache.log",
  quote = FALSE,
  row.names = FALSE
)
# At this point, put drake_cache.log under version control
# (e.g. with 'git add drake_cache.log') alongside your code.
# Now, every time you run your project, your commit history
# of hash_lot.txt is a changelog of the project's results.
# It shows which targets and imports changed on every commit.
# It is extremely difficult to track your results this way
# by putting the raw '.drake/' cache itself under version control.
})

## End(Not run)
```

---

drake_cache_log_file *Generate a flat text log file to represent the state of the cache.*

Description

This functionality is like `make(..., cache_log_file = TRUE)`, but separated and more customizable. The `drake_cache_log_file` function writes a flat text file to represents the state of all the targets and imports in the cache. If you call it after each `make()` and put the log file under version control, you can track the changes to your results over time. This way, your data is versioned alongside your code in a easy-to-view format. Hopefully, this functionality is a step toward better data versioning tools.

Usage

```
drake_cache_log_file(file = "drake_cache.log", path = getwd(),
  search = TRUE, cache = drake::get_cache(path = path, search = search,
  verbose = verbose), verbose = TRUE, jobs = 1, targets_only = FALSE)
```

Arguments

file	character scalar, name of the flat text log file.
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	drake cache. See new_cache(). If supplied, path and search are ignored.
verbose	whether to print console messages
jobs	number of jobs/workers for parallel processing
targets_only	logical, whether to output information only on the targets in your workflow plan data frame. If targets_only is FALSE, the output will include the hashes of both targets and imports.

Value

There is no return value, but a log file is generated.

See Also

drake_cache_log, make, get_cache, default_long_hash_algo, short_hash, long_hash

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# Load drake's canonical example.
load_basic_example() # Get the code with drake_example()
# Run the project and save a flat text log file.
make(my_plan)
drake_cache_log_file() # writes drake_cache.log
# The above 2 lines are equivalent to make(my_plan, cache_log_file = TRUE) # nolint
# At this point, put drake_cache.log under version control
# (e.g. with 'git add drake_cache.log') alongside your code.
# Now, every time you run your project, your commit history
# of hash_lot.txt is a changelog of the project's results.
# It shows which targets and imports changed on every commit.
# It is extremely difficult to track your results this way
# by putting the raw '.drake/' cache itself under version control.
})

## End(Not run)
```

drake_config *Create the internal runtime parameter list used internally in* make().

Description

This configuration list is also required for functions such as outdated() and vis_drake_graph(). It is meant to be specific to a single call to make(), and you should not modify it by hand afterwards. If you later plan to call make() with different arguments (especially targets), you should refresh the config list with another call to drake_config(). For changes to the targets argument specifically, it is important to recompute the config list to make sure the internal workflow network has all the targets you need. Modifying the targets element afterwards will have no effect and it could lead to false negative results from outdated()

Usage

```
drake_config(plan = drake_plan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = 1, hook = default_hook,
  cache = drake::get_cache(verbose = verbose, force = force),
  fetch_cache = NULL, parallelism = drake::default_parallelism(),
  jobs = 1, packages = rev(.packages()), prework = character(0),
  prepend = character(0), command = drake::default_Makefile_command(),
  args = drake::default_Makefile_args(jobs = jobs, verbose = verbose),
  recipe_command = drake::default_recipe_command(), timeout = Inf,
  cpu = timeout, elapsed = timeout, retries = 0, force = FALSE,
  log_progress = FALSE, graph = NULL, trigger = drake::default_trigger(),
  imports_only = FALSE, skip_imports = FALSE, skip_safety_checks = FALSE,
  lazy_load = FALSE, session_info = TRUE, cache_log_file = NULL)
```

Arguments

plan	same as for make
targets	same as for make
envir	same as for make
verbose	same as for make
hook	same as for make
cache	same as for make
fetch_cache	same as for make
parallelism	same as for make
jobs	same as for make
packages	same as for make
prework	same as for make
prepend	same as for make
command	same as for make

args	same as for make
recipe_command	same as for make
timeout	same as for make
cpu	same as for make
elapsed	same as for make
retries	same as for make
force	same as for make
log_progress	logical, whether to clear the cached progress of the targets readable by
graph	igraph object representing the workflow plan network. Overrides skip_imports.
trigger	same as for make
imports_only	logical, whether to skip building the targets in plan and just import objects and files.
skip_imports	logical, whether to totally neglect to process the imports and jump straight to the targets. This can be useful if your imports are massive and you just want to test your project, but it is bad practice for reproducible data analysis. This argument is overridden if you supply your own graph argument.
skip_safety_checks	logical, whether to skip the safety checks on your workflow to save time. Use at your own peril.
lazy_load	same as for make
session_info	same as for make
cache_log_file	same as for make

Value

The master internal configuration list of a project.

See Also

make_with_config, make, drake_plan, vis_drake_graph

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Construct the master internal configuration list.
con <- drake_config(my_plan)
# These functions are faster than otherwise
# because they use the configuration list.
outdated(config = con) # Which targets are out of date?
missed(config = con) # Which imports are missing?
# In make(..., jobs = n), it would be silly to set `n` higher than this:
max_useful_jobs(config = con)
# Show a visNetwork graph
vis_drake_graph(config = con)
```

```
# Get the underlying node/edge data frames of the graph.
dataframes_graph(config = con)
})

## End(Not run)
```

`drake_example` *Save the source code files of a drake example.*

Description

Copy a folder of code files for a drake example to the current working directory. Call `drake_example('basic')` to generate the code files from the quickstart vignette: `vignette('quickstart')`. To see the names of all the examples, run `drake_examples`.

Usage

```
drake_example(example = drake::drake_examples(), to = getwd(),
  destination = NULL, overwrite = FALSE)
```

Arguments

`example`	name of the example. To see all the available example names, run `drake_examples`.
`to`	Character scalar, file path, where to write the folder containing the code files for the example.
`destination`	Deprecated, use `to` instead.
`overwrite`	Logical, whether to overwrite an existing folder with the same name as the drake example.

Value

`NULL`

See Also

`drake_examples`, `make`, `shell_file`, `drake_batchtools_tmpl_file`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
drake_examples() # List all the drake examples.
# Sets up the same example as the quickstart vignette.
drake_example("basic")
# Sets up the SLURM example.
drake_example("slurm")
})

## End(Not run)
```

drake_examples *List the names of all the drake examples.*

Description

The 'basic' example is the one from the quickstart vignette: vignette('quickstart'). All are in the inst/examples/ folder of the package source code.

Usage

```
drake_examples()
```

Value

Names of all the drake examples.

See Also

drake_example, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
drake_examples() # List all the drake examples.
# Sets up the same example as the quickstart vignette.
drake_example("basic")
# Sets up the SLURM example.
drake_example("slurm")
})

## End(Not run)
```

drake_gc *Do garbage collection on the drake cache.*

Description

The cache is a key-value store. By default, the clean() function removes values, but not keys. Garbage collection removes the remaining dangling files.

Usage

```
drake_gc(path = getwd(), search = TRUE, verbose = 1, cache = NULL,
  force = FALSE)
```

Arguments

path	file path to the folder containing the cache. Yes, this is the parent directory containing the cache, not the cache itself, and it assumes the cache is in the '.drake' folder. If you are looking for a different cache with a known folder different from '.drake', use the this_cache() function.
search	logical, whether to search back in the file system for the cache.
verbose	logical, whether to print the location of the cache
cache	the drake/storr cache object itself, if available.
force	logical, whether to load the cache despite any back compatibility issues with the running version of drake.

Value

NULL

See Also

clean

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
# At this point, check the size of the '.drake/' cache folder.
# Clean without garbage collection.
clean(garbage_collection = FALSE)
# The '.drake/' cache folder is still about the same size.
drake_gc() # Do garbage collection on the cache.
# The '.drake/' cache folder should have gotten much smaller.
})

## End(Not run)
```

---

drake_meta *Compute the metadata of a target or import.*

Description

The metadata helps determine if the target is up to date or outdated. The metadata of imports is used to compute the metadata of targets.

Usage

```
drake_meta(target, config)
```

Arguments

target	Character scalar, name of the target to get metadata.
config	Master internal configuration list produced by drake_config().

Details

Target metadata is computed with drake_meta() and then drake:::finish_meta(). This metadata corresponds to the state of the target immediately after it was built or imported in the last make() that did not skip it. The exception to this is the $missing element of the metadata, which indicates if the target/import was missing just *before* it was built.

Value

A list of metadata on a target. Does not include the file modification time if the target is a file. That piece is provided later in make() by drake:::finish_meta.

See Also

dependency_profile, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# This example is not really a user-side demonstration.
# It just walks through a dive into the internals.
# Populate your workspace and write 'report.Rmd'.
load_basic_example() # Get the code with drake_example("basic").
# Create the master internal configuration list.
config <- drake_config(my_plan)
# Optionally, compute metadata on 'small',
# including a hash/fingerprint
# of the dependencies. If meta is not supplied,
# drake_build() computes it automatically.
meta <- drake_meta(target = "small", config = config)
# Should not yet include 'small'.
cached()
# Build 'small'.
# Equivalent to just drake_build(target = "small", config = config).
drake_build(target = "small", config = config, meta = meta)
# Should now include 'small'
cached()
readd(small)
})

## End(Not run)
```

drake_palette *Show drake's color palette.*

Description

This function is used in both the console and `vis_drake_graph()` Your console must have the crayon package enabled.

Usage

```
drake_palette()
```

Details

This palette applies to console output (internal functions `console()` and `console_many_targets()`) and the node colors in `vis_drake_graph()`. So if you want to contribute improvements to the palette, please both `drake_palette()` and `visNetwork::visNetwork(nodes = legend_nodes())`

Value

There is a console message, but the actual return value is `NULL`.

Examples

```
# Show drake's color palette as text.
drake_palette()
# Show part of the palette as an interactive visNetwork graph.
# These are the nodes in the legend of vis_drake_graph().
## Not run:
visNetwork::visNetwork(nodes = legend_nodes())

## End(Not run)
```

drake_plan *Create a workflow plan data frame for the* `plan` *argument of* `make`.

Description

Turns a named collection of target/command pairs into a workflow plan data frame for `make()` and `check()`. You can give the commands as named expressions, or you can use the `list` argument to supply them as character strings.

Usage

```
drake_plan(..., list = character(0), file_targets = FALSE,
  strings_in_dots = c("filenames", "literals"))
```

Arguments

...	A collection of symbols/targets with commands assigned to them. See the examples for details.
list	A named list of targets, where the values are commands.
file_targets	logical, whether the targets should be (single-quoted) external file targets.
strings_in_dots	Character scalar, how to treat quoted character strings in the commands specified through .... Set to "filenames" to treat all these strings as external file targets/imports (single-quoted), or to "literals" to treat them all as literal strings (double-quoted). Unfortunately, because of how R deparses code, you cannot simply leave literal quotes alone in the ... argument. R will either convert all these quotes to single quotes or double quotes. Literal quotes in the list argument are left alone.

Details

A workflow plan data frame is a data frame with a target column and a command column. Targets are the objects and files that drake generates, and commands are the pieces of R code that produce them.

For file inputs and targets, drake uses single quotes. Double quotes are reserved for ordinary strings. The distinction is important because drake thinks about how files, objects, targets, etc. depend on each other. Quotes in the list argument are left alone, but R messes with quotes when it parses the free-form arguments in ..., so use the strings_in_dots argument to control the quoting in ....

Value

A data frame of targets and commands.

Examples

```
# Create example workflow plan data frames for make()
drake_plan(small = simulate(5), large = simulate(50))
# Commands can be multi-line code chunks.
small_plan <- drake_plan(
  small_target = {
    local_object <- 1 + 1
    2 + sqrt(local_object)
  }
)
small_plan
## Not run:
make(small_plan)
cached()
readd(small_target)
# local_object only applies to the code chunk.
ls() # your environment is protected (local_object not found)

## End(Not run)
# For tighter control over commands, use the `list` argument.
```

```
drake_plan(list = c(x = "1 + 1", y = "sqrt(x)"))
# This becomes important for file targets,
# which you must put in single quotes.
# (Double quotes are for string literals.)
drake_plan(data = readRDS("my_data.rds"))
drake_plan(my_file.rds = saveRDS(1+1, "my_file.rds"), file_targets = TRUE,
  strings_in_dots = "literals")
```

`drake_quotes` *Put quotes around each element of a character vector.*

Description

Quotes are important in drake. In workflow plan data frame commands, single-quoted targets denote physical files, and double-quoted strings are treated as ordinary string literals.

Usage

```
drake_quotes(x = NULL, single = FALSE)
```

Arguments

`x`	character vector or object to be coerced to character.
`single`	Add single quotes if `TRUE` and double quotes otherwise.

Value

character vector with quotes around it

See Also

`drake_unquote`, `drake_strings`

Examples

```
# Single-quote this string.
drake_quotes("abcd", single = TRUE) # "'abcd'"
# Double-quote this string.
drake_quotes("abcd") # "\"abcd\""
```

drake_session *Return the* sessionInfo() *of the last call to* make().

Description

By default, session info is saved during make() to ensure reproducibility. Your loaded packages and their versions are recorded, for example.

Usage

```
drake_session(path = getwd(), search = TRUE, cache = drake::get_cache(path
  = path, search = search, verbose = verbose), verbose = TRUE)
```

Arguments

path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.
verbose	whether to print console messages

Value

sessionInfo() of the last call to make()

See Also

diagnose, built, imported, readd, drake_plan, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
drake_session() # Retrieve the cached sessionInfo() of the last make().
})

## End(Not run)
```

drake_strings *Turn valid expressions into character strings.*

Description

This function may be useful for constructing workflow plan data frames.

Usage

```
drake_strings(...)
```

Arguments

...	unquoted symbols to turn into character strings.

Value

a character vector

See Also

drake_quotes, drake_unquote

Examples

```
# Turn symbols into strings.
drake_strings(a, b, c, d) # [1] "a" "b" "c" "d"
```

drake_tip *Output a random tip about drake.*

Description

Tips are usually related to news and usage.

Usage

```
drake_tip()
```

Value

A character scalar with a tip on how to use drake.

Examples

```
drake_tip() # Show a tip about using drake.
message(drake_tip()) # Print out a tip as a message.
```

drake_unquote *Remove leading and trailing escaped quotes from character strings.*

Description

Quotes are important in drake. In workflow plan data frame commands, single-quoted targets denote physical files, and double-quoted strings are treated as ordinary string literals.

Usage

```
drake_unquote(x = NULL, deep = FALSE)
```

Arguments

x	character vector
deep	remove all outer quotes if TRUE and only the outermost set otherwise. Single and double quotes are treated interchangeably, and matching is not checked.

Value

character vector without leading or trailing escaped quotes around the elements

See Also

drake_quotes, drake_strings

Examples

```
x <- "'abcd'"
# Remove the literal quotes around x.
drake_unquote(x) # "abcd"
```

empty_hook *A* hook *argument to* make() *for which no targets get built and no imports get processed.*

Description

This hook forces make() to essentially do nothing.

Usage

```
empty_hook(code)
```

Arguments

code	Placeholder for the code to build a target/import. For empty_hook, this code does not actually get executed.

Value

A function that you can supply to the hook argument of make().

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Run the project with the empty hook.
make(my_plan, hook = empty_hook) # Nothing gets built!
cached() # character(0) # nolint
})

## End(Not run)
```

---

evaluate_plan *Use wildcard templating to create a workflow plan data frame from a template data frame.*

Description

The commands in workflow plan data frames can have wildcard symbols that can stand for datasets, parameters, function arguments, etc. These wildcards can be evaluated over a set of possible values using evaluate_plan.

Usage

```
evaluate_plan(plan, rules = NULL, wildcard = NULL, values = NULL,
  expand = TRUE)
```

Arguments

plan	workflow plan data frame, similar to one produced by drake_plan
rules	Named list with wildcards as names and vectors of replacements as values. This is a way to evaluate multiple wildcards at once. When not NULL, rules overrules wildcard and values if not NULL.
wildcard	character scalar denoting a wildcard placeholder
values	vector of values to replace the wildcard in the drake instructions. Will be treated as a character vector. Must be the same length as plan$command if expand is TRUE.
expand	If TRUE, create a new rows in the workflow plan data frame if multiple values are assigned to a single wildcard. If FALSE, each occurrence of the wildcard is replaced with the next entry in the values vector, and the values are recycled.

Details

Specify a single wildcard with the `wildcard` and `values` arguments. In each command, the text in `wildcard` will be replaced by each value in `values` in turn. Specify multiple wildcards with the `rules` argument, which overrules `wildcard` and `values` if not `NULL`. Here, `rules` should be a list with wildcards as names and vectors of possible values as list elements.

Value

A workflow plan data frame with the wildcards evaluated.

Examples

```
# Create the part of the workflow plan for the datasets.
datasets <- drake_plan(
  small = simulate(5),
  large = simulate(50))
# Create a template workflow plan for the analyses.
methods <- drake_plan(
  regression1 = reg1(dataset__),
  regression2 = reg2(dataset__))
# Evaluate the wildcards in the template
# to produce the actual part of the workflow plan
# that encodes the analyses of the datasets.
# Create one analysis for each combination of dataset and method.
evaluate_plan(methods, wildcard = "dataset__",
  values = datasets$target)
# Only choose some combinations of dataset and analysis method.
ans <- evaluate_plan(methods, wildcard = "dataset__",
  values = datasets$target, expand = FALSE)
ans
# For the complete workflow plan, row bind the pieces together.
my_plan <- rbind(datasets, ans)
my_plan
# Workflow plans can have multiple wildcards.
# Each combination of wildcard values will be used
# Except when expand is FALSE.
x <- drake_plan(draws = rnorm(mean = Mean, sd = Sd))
evaluate_plan(x, rules = list(Mean = 1:3, Sd = c(1, 10)))
```

---

expand_plan *Create replicates of targets.*

Description

Duplicates the rows of a workflow plan data frame. Prefixes are appended to the new target names so targets still have unique names.

Usage

```
expand_plan(plan, values = NULL)
```

Arguments

plan	workflow plan data frame
values	values to expand over. These will be appended to the names of the new targets.

Value

An expanded workflow plan data frame (with replicated targets).

Examples

```
# Create the part of the workflow plan for the datasets.
datasets <- drake_plan(
  small = simulate(5),
  large = simulate(50))
# Create replicates. If you want repeat targets,
# this is convenient.
expand_plan(datasets, values = c("rep1", "rep2", "rep3"))
```

---

failed *List the targets that failed in the last call to* make().

Description

Together, functions failed and diagnose() should eliminate the strict need for ordinary error messages printed to the console.

Usage

```
failed(path = getwd(), search = TRUE, cache = drake::get_cache(path =
  path, search = search, verbose = verbose), verbose = TRUE)
```

Arguments

path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.
verbose	whether to print console messages

Value

A character vector of target names.

See Also

diagnose, session, built, imported, readd, drake_plan, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
failed() # Should show that no targets failed.
# Build a workflow plan doomed to fail:
bad_plan <- drake_plan(x = function_doesnt_exist())
try(make(bad_plan), silent = TRUE) # error
failed() # "x"
diagnose(x) # Retrieve the cached error log of x.
})

## End(Not run)
```

---

find_cache *Search up the file system for the nearest drake cache.*

Description

Only works if the cache is a file system in a hidden folder named `.drake` (default).

Usage

```
find_cache(path = getwd(),
  directory = basename(drake::default_cache_path()))
```

Arguments

`path`	starting path for search back for the cache. Should be a subdirectory of the drake project.
`directory`	Name of the folder containing the cache.

Value

File path of the nearest drake cache or `NULL` if no cache is found.

See Also

drake_plan, make,

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the target.
# Find the file path of the project's cache.
# Search up through parent directories if necessary.
find_cache()
})

## End(Not run)
```

`find_project` *Search up the file system for the nearest root path of a drake project.*

Description

Only works if the cache is a file system in a folder named `.drake` (default).

Usage

```
find_project(path = getwd())
```

Arguments

`path`	starting path for search back for the project. Should be a subdirectory of the drake project.

Value

File path of the nearest drake project or `NULL` if no drake project is found.

See Also

`drake_plan`, `make`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the target.
# Find the root directory of the current drake project.
# Search up through parent directories if necessary.
find_cache()
})

## End(Not run)
```

gather_plan *Write commands to combine several targets into one or more overarching targets.*

Description

Creates a new workflow plan data frame with a single new target. This new target is a list, vector, or other aggregate of a collection of existing targets in another workflow plan data frame.

Usage

```
gather_plan(plan = NULL, target = "target", gather = "list")
```

Arguments

plan	workflow plan data frame of prespecified targets
target	name of the new aggregated target
gather	function used to gather the targets. Should be one of list(...), c(...), rbind(...), or similar.

Value

A workflow plan data frame that aggregates multiple prespecified targets into one additional target downstream.

Examples

```
# Workflow plan for datasets:
datasets <- drake_plan(
  small = simulate(5),
  large = simulate(50))
# Create a new target that brings the datasets together.
gather_plan(datasets, target = "my_datasets")
# This time, the new target just appends the rows of 'small' and 'large'
# into a single matrix or data frame.
gathered <- gather_plan(
  datasets, target = "aggregated_data", gather = "rbind"
)
gathered
# For the complete workflow plan, row bind the pieces together.
my_plan <- rbind(datasets, gathered)
my_plan
```

get_cache *Get the drake cache, optionally searching up the file system.*

Description

Only works if the cache is in a folder called `.drake/`.

Usage

```
get_cache(path = getwd(), search = TRUE, verbose = 1, force = FALSE,
  fetch_cache = NULL)
```

Arguments

path	file path to the folder containing the cache. Yes, this is the parent directory containing the cache, not the cache itself, and it assumes the cache is in the '.drake' folder. If you are looking for a different cache with a known folder different from '.drake', use the `this_cache()` function.
search	logical, whether to search back in the file system for the cache.
verbose	logical, whether to print the location of the cache
force	logical, whether to load the cache despite any back compatibility issues with the running version of drake.
fetch_cache	character vector containing lines of code. The purpose of this code is to fetch the `storr` cache with a command like `storr_rds()` or `storr_dbi()`, but customized. This feature is experimental.

Value

A drake/storr cache in a folder called `.drake/`, if available. `NULL` otherwise.

See Also

`this_cache`, `new_cache`, `recover_cache`, `config`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
clean(destroy = TRUE)
# No cache is available.
get_cache() # NULL
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
x <- get_cache() # Now, there is a cache.
# List the objects readable from the cache with readd().
x$list() # Or x$list(namespace = x$default_namespace)
})
```

```
## End(Not run)
```

`imported` *List all the imports in the drake cache.*

Description

An import is a non-target object processed by `make()`. Targets in the workflow plan data frame (see `drake_config()` may depend on imports.

Usage

```
imported(files_only = FALSE, path = getwd(), search = TRUE,
  cache = drake::get_cache(path = path, search = search, verbose = verbose),
  verbose = TRUE, jobs = 1)
```

Arguments

`files_only`	logical, whether to show imported files only and ignore imported objects. Since all your functions and all their global variables are imported, the full list of imported objects could get really cumbersome.
`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project.
`search`	logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
`cache`	drake cache. See `new_cache()`. If supplied, `path` and `search` are ignored.
`verbose`	whether to print console messages
`jobs`	number of jobs/workers for parallel processing

Value

Character vector naming the imports in the cache.

See Also

`cached`, `loadd`, `built`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Load the canonical example.
make(my_plan) # Run the project, build the targets.
imported() # List all the imported objects/files in the cache.
# For imported files, only the fingerprints/hashes are stored.
})

## End(Not run)
```

in_progress *List the targets that either (1) are currently being built during a* make() *, or (2) were being built if the last* make() *quit unexpectedly.*

Description

Similar to progress().

Usage

```
in_progress(path = getwd(), search = TRUE, cache = drake::get_cache(path =
  path, search = search, verbose = verbose), verbose = TRUE)
```

Arguments

path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.
verbose	whether to print console messages

Value

A character vector of target names.

See Also

diagnose, session, built, imported, readd, drake_plan, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Kill before targets finish.
# If you interrupted make(), some targets will probably be listed:
in_progress()
})

## End(Not run)
```

knitr_deps *Find the drake dependencies of a dynamic knitr report target.*

Description

To enable drake to watch for the dependencies of a knitr report, the command in your workflow plan data frame must call `knitr::knit()` directly. In other words, the command must look something like `knit('your_report.Rmd')` or `knit('your_report.Rmd', quiet = TRUE)`.

Usage

```
knitr_deps(target)
```

Arguments

`target`	file path to the file or name of the file target, source text of the document.

Details

Drake looks for dependencies in the document by analyzing evaluated code chunks for other targets/imports mentioned in `loadd()` and `readd()`.

Value

A character vector of the names of dependencies.

See Also

`deps`, `make`, `load_basic_example`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
knitr_deps("'report.Rmd'") # Files must be single-quoted.
# Find the dependencies of the compiled output target, 'report.md'.
knitr_deps("report.Rmd")
make(my_plan) # Run the project.
# Work only on the Rmd source, not the output.
try(knitr_deps("'report.md'"), silent = FALSE) # error
})

## End(Not run)
```

legend_nodes *Create the nodes data frame used in the legend of* vis_drake_graph().

Description

Output a visNetwork-friendly data frame of nodes. It tells you what the colors and shapes mean in vis_drake_graph().

Usage

```
legend_nodes(font_size = 20)
```

Arguments

font_size	font size of the node label text

Value

A data frame of legend nodes for vis_drake_graph().

See Also

drake_palette(), vis_drake_graph(), dataframes_graph()

Examples

```
## Not run:
# Show the legend nodes used in vis_drake_graph().
# For example, you may want to inspect the color palette more closely.
visNetwork::visNetwork(nodes = legend_nodes())

## End(Not run)
```

loadd *Load multiple targets or imports from the drake cache.*

Description

Loads the object(s) into the current workspace (or envir if given). Defaults to loading the whole cache if arguments ... and list are not set (or all the imported objects if in addition imported_only is TRUE).

Usage

```
loadd(..., list = character(0), imported_only = FALSE, path = getwd(),
  search = TRUE, cache = drake::get_cache(path = path, search = search,
  verbose = verbose), namespace = NULL, envir = parent.frame(), jobs = 1,
  verbose = 1, deps = FALSE, lazy = FALSE)
```

Arguments

`...`	targets to load from the cache, as names (unquoted) or character strings (quoted). Similar to `...` in `remove(...)`.
`list`	character vector naming targets to be loaded from the cache. Similar to the `list` argument of `remove()`.
`imported_only`	logical, whether only imported objects should be loaded.
`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project.
`search`	logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
`cache`	optional drake cache. See codenew_cache(). If `cache` is supplied, the `path` and `search` arguments are ignored.
`namespace`	character scalar, name of an optional storr namespace to load from.
`envir`	environment to load objects into. Defaults to the calling environment (current workspace).
`jobs`	number of parallel jobs for loading objects. On non-Windows systems, the loading process for multiple objects can be lightly parallelized via `parallel::mclapply()`. just set jobs to be an integer greater than 1. On Windows, `jobs` is automatically demoted to 1.
`verbose`	logical, whether to print console messages
`deps`	logical, whether to load any cached dependencies of the targets instead of the targets themselves. This is useful if you know your target failed and you want to debug the command in an interactive session with the dependencies in your workspace. One caveat: to find the dependencies, `loadd()` uses information that was stored in a `drake_config()` list and cached during the last `make()`. That means you need to have already called `make()` if you set `deps` to `TRUE`.
`lazy`	logical, whether to lazy load with `delayedAssign()` rather than the more eager `assign()`.

Value

`NULL`

See Also

`cached`, `built`, `imported`, `drake_plan`, `make`,

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the projects, build the targets.
loadd(small) # Load target 'small' into your workspace.
small
# For many targets, you can parallelize loadd()
# using the 'jobs' argument.
loadd(list = c("small", "large"), jobs = 2)
# Load the dependencies of the target, coef_regression2_small
loadd(coef_regression2_small, deps = TRUE)
# Load all the imported objects/functions.
loadd(imported_only = TRUE)
# Load everything, including built targets.
# Be sure your computer has enough memory.
loadd()
})

## End(Not run)
```

`load_basic_example` *Load the basic example from* `drake_example("basic")`

Description

Is there an association between the weight and the fuel efficiency of cars? To find out, we use the mtcars dataset. The mtcars dataset itself only has 32 rows, so we generate two larger bootstrapped datasets and then analyze them with regression models. Finally, we summarize the regression models to see if there is an association.

Usage

```
load_basic_example(envir = parent.frame(), cache = NULL,
  report_file = "report.Rmd", overwrite = FALSE, to = report_file,
  verbose = TRUE, force = FALSE)
```

Arguments

`envir`	The environment to load the example into. Defaults to your workspace. For an insulated workspace, set `envir = new.env(parent = globalenv())`.
`cache`	Optional `storr` cache to use.
`report_file`	where to write the report file `report.Rmd`.
`overwrite`	logical, whether to overwrite an existing file `report.Rmd`
`to`	deprecated, where to write the dynamic report source file `report.Rmd`
`verbose`	logical, whether to print console messages.
`force`	logical, whether to force the loading of a non-back-compatible cache from a previous version of drake.

Details

Use drake_example('basic') to get the code for the basic example. The included R script is a detailed, heavily-commented walkthrough. The quickstart vignette at https://github.com/ropensci/drake/blob/master/vignettes/quickstart.Rmd # nolint and https://ropensci.github.io/drake/articles/quickstart.html also walks through the basic example. This function also writes/overwrites the file, report.Rmd.

Value

A drake_config() configuration list.

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# Populate your workspace and write 'report.Rmd'.
load_basic_example() # Get the code: drake_example("basic")
# Check the dependencies of an imported function.
deps(reg1)
# Check the dependencies of commands in the workflow plan.
deps(my_plan$command[1])
deps(my_plan$command[4])
# Plot the interactive network visualization of the workflow.
config <- drake_config(my_plan)
vis_drake_graph(config)
# Run the workflow to build all the targets in the plan.
make(my_plan)
# For the reg2() model on the small dataset,
# the p-value is so small that there may be an association
# between weight and fuel efficiency after all.
readd(coef_regression2_small)
# Remove the whole cache.
clean(destroy = TRUE)
# Clean up the imported file.
unlink('report.Rmd')
})

## End(Not run)
```

---

long_hash *Get the long hash algorithm of a drake cache.*

Description

See the advanced storage tutorial at https://ropensci.github.io/drake/articles/storage.html for details.

Usage

```
long_hash(cache = drake::get_cache(verbose = verbose), verbose = TRUE)
```

Arguments

cache	drake cache
verbose	whether to print console messages

Value

A character vector naming a hash algorithm.

See Also

default_short_hash_algo, default_long_hash_algo

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Run the project and return the internal master configuration list.
config <- make(my_plan)
# Locate the storr cache.
cache <- config$cache
# Get the long hash algorithm of the cache.
long_hash(cache)
})

## End(Not run)
```

---

make *Run your project (build the outdated targets).*

Description

This is the central, most important function of the drake package. It runs all the steps of your workflow in the correct order, skipping any work that is already up to date. See https://ropensci.github.io/drake/ for the full documentation, which includes multiple in-depth tutorials.

Usage

```
make(plan = drake_plan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = 1, hook = default_hook,
  cache = drake::get_cache(verbose = verbose, force = force),
  fetch_cache = NULL, parallelism = drake::default_parallelism(),
  jobs = 1, packages = rev(.packages()), prework = character(0),
  prepend = character(0), command = drake::default_Makefile_command(),
  args = drake::default_Makefile_args(jobs = jobs, verbose = verbose),
  recipe_command = drake::default_recipe_command(), log_progress = TRUE,
  imports_only = FALSE, timeout = Inf, cpu = NULL, elapsed = NULL,
  retries = 0, force = FALSE, return_config = NULL, graph = NULL,
```

```
  trigger = drake::default_trigger(), skip_imports = FALSE,
  skip_safety_checks = FALSE, config = NULL, lazy_load = FALSE,
  session_info = TRUE, cache_log_file = NULL)
```

Arguments

plan	workflow plan data frame. A workflow plan data frame is a data frame with a `target` column and a `command` column. Targets are the objects and files that drake generates, and commands are the pieces of R code that produce them. Use the function `drake_plan()` to generate workflow plan data frames easily, and see functions `analyses()`, `summaries()`, `evaluate()`, `expand()`, and `gather()` for easy ways to generate large workflow plan data frames.
targets	character string, names of targets to build. Dependencies are built too. Together, the `plan` and `targets` comprise the workflow network (i.e. the `graph` argument). Changing either will change the network.
envir	environment to use. Defaults to the current workspace, so you should not need to worry about this most of the time. A deep copy of `envir` is made, so you don't need to worry about your workspace being modified by `make`. The deep copy inherits from the global environment. Wherever necessary, objects and functions are imported from `envir` and the global environment and then reproducibly tracked as dependencies.
verbose	logical or numeric, control printing to the console. **0 or** `FALSE`**:** print nothing. **1 or** `TRUE`**:** print checks and targets to build. **2:** print checks, targets, to build, and any potentially missing items. **3:** full verbosity: print checks, targets to build, potentially missing items, and imports.
hook	function with at least one argument. The hook is as a wrapper around the code that drake uses to build a target (see the body of `drake:::build_in_hook()`). Hooks can control the side effects of build behavior. For example, to redirect output and error messages to text files, you might use the built-in `silencer_hook()`, as in `make(my_plan, hook = silencer_hook)`. The silencer hook is useful for distributed parallelism, where the calling R process does not have control over all the error and output streams. See also `output_sink_hook()` and `message_sink_hook()`. For your own custom hooks, treat the first argument as the code that builds a target, and make sure this argument is actually evaluated. Otherwise, the code will not run and none of your targets will build. For example, `function(code){force(code)}` is a good hook and `function(code){message("Avoiding the co` is a bad hook.
cache	drake cache as created by `new_cache()`. See also `get_cache()`, `this_cache()`, and `recover_cache()`
fetch_cache	character vector containing lines of code. The purpose of this code is to fetch the `storr` cache with a command like `storr_rds()` or `storr_dbi()`, but customized. This feature is experimental. It will turn out to be necessary if you are using both custom non-RDS caches and distributed parallelism (`parallelism = "future_lapply"` or `"Makefile"`) because the distributed R sessions need to know how to load the cache.

parallelism — character, type of parallelism to use. To list the options, call parallelism_choices(). For detailed explanations, see ?parallelism_choices, the tutorial vignettes, or the tutorial files generated by drake_example("basic")

jobs — number of parallel processes or jobs to run. See max_useful_jobs() or vis_drake_graph() to help figure out what the number of jobs should be. Windows users should not set jobs > 1 if parallelism is "mclapply" because mclapply() is based on forking. Windows users who use parallelism == "Makefile" will need to download and install Rtools.

For "future_lapply" parallelism, jobs only applies to the imports. To set the max number of jobs for "future_lapply" parallelism, set the workers argument where it exists: for example, call future::plan(multisession(workers = 4)), then call make(your_plan, parallelism = "future_lapply"). You might also try options(mc.cores = jobs), or see ?future::future::.options for environment variables that set the max number of jobs.

If parallelism is "Makefile", Makefile-level parallelism is only used for targets in your workflow plan data frame, not imports. To process imported objects and files, drake selects the best parallel backend for your system and uses the number of jobs you give to the jobs argument to make(). To use at most 2 jobs for imports and at most 4 jobs for targets, run make(..., parallelism = "Makefile", jobs = 2, args

packages — character vector packages to load, in the order they should be loaded. Defaults to rev(.packages()), so you should not usually need to set this manually. Just call library() to load your packages before make(). However, sometimes packages need to be strictly forced to load in a certain order, especially if parallelism is "Makefile". To do this, do not use library() or require() or loadNamespace() or attachNamespace() to load any libraries beforehand. Just list your packages in the packages argument in the order you want them to be loaded. If parallelism is "mclapply", the necessary packages are loaded once before any targets are built. If parallelism is "Makefile", the necessary packages are loaded once on initialization and then once again for each target right before that target is built.

prework — character vector of lines of code to run before build time. This code can be used to load packages, set options, etc., although the packages in the packages argument are loaded before any prework is done. If parallelism is "mclapply", the prework is run once before any targets are built. If parallelism is "Makefile", the prework is run once on initialization and then once again for each target right before that target is built.

prepend — lines to prepend to the Makefile if parallelism is "Makefile". See the vignettes (vignette(package = "drake")) to learn how to use prepend to take advantage of multiple nodes of a supercomputer.

command — character scalar, command to call the Makefile generated for distributed computing. Only applies when parallelism is "Makefile". Defaults to the usual "make" (default_Makefile_command()), but it could also be "lsmake" on supporting systems, for example. command and args are executed via system2(command, args) to run the Makefile. If args has something like "--jobs=2", or if jobs >= 2 and args is left alone, targets will be distributed over independent parallel R sessions wherever possible.

args	command line arguments to call the Makefile for distributed computing. For advanced users only. If set, jobs and verbose are overwritten as they apply to the Makefile. command and args are executed via system2(command, args) to run the Makefile. If args has something like "--jobs=2", or if jobs >= 2 and args is left alone, targets will be distributed over independent parallel R sessions wherever possible.
recipe_command	Character scalar, command for the Makefile recipe for each target.
log_progress	logical, whether to log the progress of individual targets as they are being built. Progress logging creates a lot of little files in the cache, and it may make builds a tiny bit slower. So you may see gains in storage efficiency and speed with make(..., log_progress = FALSE). But be warned that progress() and in_progress() will no longer work if you do that.
imports_only	logical, whether to skip building the targets in plan and just import objects and files.
timeout	Seconds of overall time to allow before imposing a timeout on a target. Passed to R.utils::withTimeout(). Assign target-level timeout times with an optional timeout column in plan.
cpu	Seconds of cpu time to allow before imposing a timeout on a target. Passed to R.utils::withTimeout(). Assign target-level cpu timeout times with an optional cpu column in plan.
elapsed	Seconds of elapsed time to allow before imposing a timeout on a target. Passed to R.utils::withTimeout(). Assign target-level elapsed timeout times with an optional elapsed column in plan.
retries	Number of retries to execute if the target fails. Assign target-level retries with an optional retries column in plan.
force	Force make() to build your targets even if some about your setup is not quite right: for example, if you are using a version of drake that is not back compatible with your project's cache.
return_config	Logical, whether to return the internal list of runtime configuration parameters used by make(). This argument is deprecated. Now, a configuration list is always invisibly returned.
graph	An igraph object from the previous make(). Supplying a pre-built graph could save time. The graph is constructed by build_drake_graph(). You can also get one from config(my_plan)$graph. Overrides skip_imports.
trigger	Name of the trigger to apply to all targets. Ignored if plan has a trigger column. Must be in triggers(). See ?triggers for explanations of the choices.
skip_imports	logical, whether to totally neglect to process the imports and jump straight to the targets. This can be useful if your imports are massive and you just want to test your project, but it is bad practice for reproducible data analysis. This argument is overridden if you supply your own graph argument.
skip_safety_checks	logical, whether to skip the safety checks on your workflow. Use at your own peril.
config	optional master configuration list created by drake_config(). Using one could cut out some superfluous overhead. Overrides all other arguments if supplied

lazy_load — logical. Should always be set to FALSE for "parLapply" parallelism and jobs greater than 1. For local multi-session parallelism and lazy loading, try library(future); future::pla and then make(..., parallelism = "future_lapply", lazy_load = TRUE). If lazy_load is FALSE, drake prunes the execution environment before every parallelizable stages, removing all superfluous targets and then loading any dependencies it will need for the targets in the current parallelizable stage. In other words, drake prepares the environment in advance for all the whole collection of targets in the stage. If lazy_load is TRUE, drake assigns promises to load any dependencies at the last minute. Lazy loading may be more memory efficient in some use cases, but it may duplicate the loading of dependencies, costing time.

session_info — logical, whether to save the sessionInfo() to the cache. This behavior is recommended for serious make()s for the sake of reproducibility. This argument only exists to speed up tests. Apparently, sessionInfo() is a bottleneck for small make()s.

cache_log_file — Name of the cache log file to write. If TRUE, the default file name is used (drake_cache.log). If NULL, no file is written. If activated, this option uses drake_cache_log_file() to write a flat text file to represent the state of the cache (fingerprints of all the targets and imports). If you put the log file under version control, your commit history will give you an easy representation of how your results change over time as the rest of your project changes. Hopefully, this is a step in the right direction for data reproducibility.

Value

The master internal configuration list, mostly containing arguments to make() and important objects constructed along the way. See config() for more details.

See Also

make_with_config, drake_plan, drake_plan, vis_drake_graph, max_useful_jobs, shell_file, default_hook, silencer_hook, triggers

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
config <- drake_config(my_plan)
outdated(config) # Which targets need to be (re)built?
my_jobs = max_useful_jobs(config) # Depends on what is up to date.
make(my_plan, jobs = 2) # Build what needs to be built.
outdated(config) # Everything is up to date.
# Change one of your imported function dependencies.
reg2 = function(d){
  d$x3 = d$x^3
  lm(y ~ x3, data = d)
}
outdated(config) # Some targets depend on reg2().
vis_drake_graph(config) # See how they fit in an interactive graph.
make(my_plan) # Rebuild just the outdated targets.
```

```
outdated(config) # Everything is up to date again.
make(my_plan, cache_log_file = TRUE) # Write a text log file this time.
vis_drake_graph(config) # The colors changed in the graph.
clean() # Start from scratch.
# Rerun with "Makefile" parallelism with at most 4 jobs.
# Requires Rtools on Windows.
# make(my_plan, parallelism = "Makefile", jobs = 4) # nolint
clean() # Start from scratch.
# Specify your own Makefile recipe.
# Requires Rtools on Windows.
# make(my_plan, parallelism = "Makefile", jobs = 4, # nolint
#   recipe_command = "R -q -e") # nolint
})

## End(Not run)
```

`Makefile_recipe` *For* `make(..., parallelism = "Makefile")`*, see what your* `Makefile` *recipes will look like in advance.*

Description

Relevant to `"Makefile"` parallelism only.

Usage

```
Makefile_recipe(recipe_command = drake::default_recipe_command(),
  target = "your_target", cache_path = drake::default_cache_path())
```

Arguments

`recipe_command`	The Makefile recipe command. See `default_recipe_command()`.
`target`	character scalar, name of your target
`cache_path`	path to the drake cache. In practice, this defaults to the hidden `.drake/` folder, but this can be customized. In the Makefile, the drake cache is coded with the Unix variable 'DRAKE_CACHE' and then dereferenced with '$(DRAKE_CACHE)'. To simplify things for users who may be unfamiliar with Unix variables, the `recipe()` function just shows the literal path to the cache.

Details

Makefile recipes to build targets are customizable. Use the `Makefile_recipe()` function to show and tweak Makefile recipes in advance, and see `default_recipe_command()` and `r_recipe_wildcard()` for more clues. The default recipe is `Rscript -e 'R_RECIPE'`, where `R_RECIPE` is the wildcard for the recipe in R for making the target. In writing the Makefile, `R_RECIPE` is replaced with something like `drake::mk("name_of_target", "path_to_cache")`. So when you call `make(..., parallelism = "Makefile", recipe_command = "R -e 'R_RECIPE' -q")`, # nolint from within R, the `Makefile` builds each target with the `Makefile` recipe, `R -e 'drake::mk("this_target", "path_to`

But since `R -q -e` fails on Windows, so the default `recipe_command` argument is `"Rscript -e 'R_RECIPE'"` (equivalently just `"Rscript -e"`), so the default `Makefile` recipe for each target is `Rscript -e 'drake::mk("this_target`

Value

A character scalar with a Makefile recipe.

See Also

`default_recipe_command`, `r_recipe_wildcard`, `make`

Examples

```
# Only relevant for "Makefile" parallelism:
Makefile_recipe() # Show an example Makefile recipe.
# Customize your Makefile recipe.
Makefile_recipe(
  target = "this_target",
  recipe_command = "R -e 'R_RECIPE' -q",
  cache_path = "custom_cache"
)
default_recipe_command() # "Rscript -e 'R_RECIPE'" # nolint
r_recipe_wildcard() # "R_RECIPE"
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Look at the Makefile generated by the following.
# make(my_plan, paralleliem = "Makefile") # Requires Rtools on Windows. # nolint
# Generates a Makefile with "R -q -e" rather than
# "Rscript -e".
# Be aware the R -q -e fails on Windows.
# make(my_plan, parallelism = "Makefile", jobs = 2, # nolint
#   recipe_command = "R -q -e") # nolint
# Same thing:
clean() # Start from scratch.
# make(my_plan, parallelism = "Makefile", jobs = 2, # nolint
#   recipe_command = "R -q -e 'R_RECIPE'") # nolint
clean() # Start from scratch.
# make(my_plan, parallelism = "Makefile", jobs = 2, # nolint
#   recipe_command = "R -e 'R_RECIPE' -q") # nolint
})

## End(Not run)
```

`make_imports` *Just make the imports.*

Description

make() is the central, most important function of the drake package. make() runs all the steps of your workflow in the correct order, skipping any work that is already up to date. During make(), there are two kinds of processing steps: "imports", which are pre-existing functions and input data files that are loaded or checked, and targets, which are serious reproducibly-tracked data analysis steps that have commands in your workflow plan data frame. The make_targets() function just makes the targets (skipping any targets that are already up to date) and make_imports() just makes the imports. Most users should just use make() instead of either make_imports() or make_targets(). See https://ropensci.github.io/drake/ for the full documentation of drake, including multiple in-depth tutorials.

Usage

```
make_imports(config)
```

Arguments

config	a configuration list returned by config()

Value

The master internal configuration list used by make().

See Also

make, config, make_targets

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Generate the master internal configuration list.
con <- drake_config(my_plan)
# Just cache the imports, do not build any targets.
make_imports(config = con)
# Just make the targets
make_targets(config = con)
})

## End(Not run)
```

make_targets *Just build the targets.*

Description

`make()` is the central, most important function of the drake package. `make()` runs all the steps of your workflow in the correct order, skipping any work that is already up to date. During `make()`, there are two kinds of processing steps: "imports", which are pre-existing functions and input data files that are loaded or checked, and targets, which are serious reproducibly-tracked data analysis steps that have commands in your workflow plan data frame. The `make_targets()` function just makes the targets (skipping any targets that are already up to date) and `make_imports()` just makes the imports. Most users should just use `make()` instead of either `make_imports()` or `make_targets()`. See https://ropensci.github.io/drake/ for the full documentation of drake, including multiple in-depth tutorials.

Usage

```
make_targets(config)
```

Arguments

config	a configuration list returned by `config()`

Value

The master internal configuration list used by `make()`.

See Also

`make`, `config`, `make_imports`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Generate the master internal configuration list.
con <- drake_config(my_plan)
# Just cache the imports, do not build any targets.
make_imports(config = con)
# Just make the targets
make_targets(config = con)
})

## End(Not run)
```

make_with_config *Run* make(), *on an existing internal configuration list.*

Description

Use drake_config() to create the config argument.

Usage

```
make_with_config(config)
```

Arguments

config	An input internal configuration list

Value

An output internal configuration list

See Also

make, drake_config

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# The following lines are the same as make(my_plan)
config <- drake_config(my_plan) # Create the internal config list.
make_with_config(config = config) # Run the project, build the targets.
})

## End(Not run)
```

max_useful_jobs *Suggest an upper bound on the jobs in the next call to* make(..., jobs = YOUR_CHOICE).

Description

The best number of jobs should be somewhere between 1 and max_useful_jobs(...). Any additional jobs more than max_useful_jobs(...) should be superfluous, and could even slow you down for make(..., parallelism = 'parLapply').

Usage

```
max_useful_jobs(config, imports = c("files", "all", "none"),
  from_scratch = FALSE)
```

Arguments

config	internal configuration list of make(...), produced also with drake_config().
imports	Set the imports argument to change your assumptions about how fast objects/files are imported. Possible values: • 'all': Factor all imported files/objects into calculating the max useful number of jobs. Note: this is not appropriate for make(.., parallelism = 'Makefile') because imports are processed sequentially for the Makefile option. • 'files': Factor all imported files into the calculation, but ignore all the other imports. • 'none': Ignore all the imports and just focus on the max number of useful jobs for parallelizing targets.
from_scratch	logical, whether to assume the next make() will run from scratch so that all targets are attempted.

Details

Set the imports argument to change your assumptions about how fast objects/files are imported. IMPORTANT: you must be in the root directory of your project.

Value

A numeric scalar, the maximum number of useful jobs for make(..., jobs = ...).

See Also

vis_drake_graph, build_drake_graph, shell_file, drake_config()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
config <- drake_config(my_plan) # Standard drake configuration list.
# Look at the graph. The work proceeds column by column
# in parallelizable stages. The maximum number of useful jobs
# is determined by the number and kind of targets/imports
# in the columns.
vis_drake_graph(config = config)
# Should be 8 because everythign is out of date.
max_useful_jobs(config = config) # 8
# Take into account targets and imported files.
max_useful_jobs(config = config, imports = 'files') # 8
# Include imported R objects too.
max_useful_jobs(config = config, imports = 'all') # 9
```

```
# Exclude all imported objects.
max_useful_jobs(config = config, imports = 'none') # 8
config <- make(my_plan) # Run the project, build the targets.
vis_drake_graph(config = config) # Everything is up to date.
# Ignore the targets already built.
max_useful_jobs(config = config) # 1
max_useful_jobs(config = config, imports = 'files') # 1
# Imports are never really skipped in make().
max_useful_jobs(config = config, imports = 'all') # 9
max_useful_jobs(config = config, imports = 'none') # 0
# Change a function so some targets are now out of date.
reg2 = function(d){
  d$x3 = d$x^3
  lm(y ~ x3, data = d)
}
vis_drake_graph(config = config)
# We have a different number for max useful jobs.
max_useful_jobs(config = config) # 4
# By default, max_useful_jobs() takes into account which
# targets are out of date. To assume you are building from scratch,
# consider using the "always" trigger.
max_useful_jobs(config, from_scratch = TRUE, imports = 'files') # 8
max_useful_jobs(config, from_scratch = TRUE, imports = 'all') # 9
max_useful_jobs(config, from_scratch = TRUE, imports = 'none') # 8
})

## End(Not run)
```

`message_sink_hook` *An example* hook *argument to* `make()` *that redirects error messages to files.*

Description

Most users do not need to micromanage hooks.

Usage

```
message_sink_hook(code)
```

Arguments

code	code to run to build the target.

Value

A function that you can supply to the hook argument of `make()`.

See Also

make, silencer_hook, output_sink_hook

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# Test out the message sink hook on its own.
try(
  message_sink_hook({
    cat(1234)
    stop(5678)
  }),
  silent = FALSE
)
# Create a new workflow plan.
x <- drake_plan(loud = cat(1234), bad = stop(5678))
# Run the project. All messages should be suppressed.
try(make(x, hook = message_sink_hook), silent = FALSE)
})

## End(Not run)
```

---

migrate_drake_project *Reconfigure an old project (built with drake <= 4.4.0) to be compatible with later versions of drake.*

Description

Migrate a project/cache from drake 4.4.0 or earlier to be compatible with the version of drake on your system.

Usage

```
migrate_drake_project(path = drake::default_cache_path(), jobs = 1)
```

Arguments

path	Full path to the cache
jobs	number of jobs for light parallelism. (Disabled on Windows.)

Details

Drake versions after 4.4.0 have a different internal structure for the cache. This means projects built with drake 4.4.0 or before are not compatible with projects built with a later version of drake. The migrate_drake_project() function converts an old cache to a format compatible with the version of drake installed on your system. Important note: build times and other non-essential metadata are lost during migration. A migration is successful if the transition preserves target

status: that is, outdated targets remain outdated and up to date targets remain up to date. At the end, `migrate_drake_project()` tells you whether the migration is successful. If it is not successful, `migrate_drake_project()` tells you where it backed up your old project.

Value

`TRUE` if the migration was successful, `FALSE` otherwise. A migration is successful if the transition preserves target status: that is, outdated targets remain outdated and up to date targets remain up to date.

See Also

`rescue_cache`, `make`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# With drake 4.3.0:
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the old project.
# Now, install drake >= 5.0.0
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Error: cache is not back compatible.
# Convert the project's '.drake/' cache to the new format.
migrate_drake_project()
make(my_plan) # Everything is still up to date!
# Outdated objects from before migration
# should remain out of date afterwards.
config <- drake_config(my_plan)
outdated(config)
})

## End(Not run)
```

---

missed *Report any import objects required by your drake_plan plan but missing from your workspace.*

Description

Checks your workspace/environment and file system.

Usage

```
missed(config)
```

Arguments

config	internal runtime parameter list of make(...), produced by both drake_config() and make().

Value

Character vector of names of missing objects and files.

See Also

outdated

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
config <- load_basic_example() # Get the code with drake_example("basic").
missed(config) # All the imported files and objects should be present.
rm(reg1) # Remove an import dependency from you workspace.
missed(config) # Should report that reg1 is missing.
})

## End(Not run)
```

---

new_cache *Make a new* drake *cache.*

Description

Uses the storr_rds() function from the storr package.

Usage

```
new_cache(path = drake::default_cache_path(), verbose = 1, type = NULL,
  short_hash_algo = drake::default_short_hash_algo(),
  long_hash_algo = drake::default_long_hash_algo(), ...)
```

Arguments

path	file path to the cache if the cache is a file system cache.
verbose	logical, whether to print out the path of the cache.
type	deprecated argument. Once stood for cache type. Use storr to customize your caches instead.
short_hash_algo	short hash algorithm for the cache. See default_short_hash_algo() and make()
long_hash_algo	long hash algorithm for the cache. See default_long_hash_algo() and make()
...	other arguments to the cache constructor

Value

A newly created drake cache as a storr object.

See Also

default_short_hash_algo, default_long_hash_algo, make

Examples

```
## Not run:
test_with_dir("Quarantine new_cache() side effects.", {
clean(destroy = TRUE) # Should not be necessary.
unlink("not_hidden", recursive = TRUE) # Should not be necessary.
cache1 <- new_cache() # Creates a new hidden '.drake' folder.
cache2 <- new_cache(path = "not_hidden", short_hash_algo = "md5")
clean(destroy = TRUE, cache = cache2)
})

## End(Not run)
```

---

next_stage *List the targets that will be built in the first parallelizable stage of the next call to* make.

Description

Similar to the first stage in the output of parallel_stages().

Usage

```
next_stage(config)
```

Arguments

config	A master configuration list produced by drake_config() or make()

Value

A character vector of the targets to be made in the first parallel stage of the next call to make.

See Also

parallel_stages, make, drake_config

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
config <- load_basic_example() # Get the code with drake_example("basic").
next_stage(config = config)    # "small" and "large"
})

## End(Not run)
```

`outdated` *List the targets that are out of date.*

Description

Outdated targets will be rebuilt in the next `make()`.

Usage

```
outdated(config, make_imports = TRUE)
```

Arguments

`config`	option internal runtime parameter list of `make(...)`, produced with `drake_config()`. You must use a fresh `config` argument with an up-to-date `config$targets` element that was never modified by hand. If needed, rerun `drake_config()` early and often. See the details in the help file for `drake_config()`.
`make_imports`	logical, whether to make the imports first. Set to `FALSE` to save some time and risk obsolete output.

Details

`outdated()` is sensitive to the alternative triggers described at https://github.com/ropensci/drake/blob/master/vignettes/debug.Rmd#test-with-triggers. # nolint For example, even if `outdated(...)` shows everything up to date, `outdated(..., trigger = "always")` will show all targets out of date. You must use a fresh `config` argument with an up-to-date `config$targets` element that was never modified by hand. If needed, rerun `drake_config()` early and often. See the details in the help file for `drake_config()`.

Value

Character vector of the names of outdated targets.

See Also

`missed`, `drake_plan`, `make`, `vis_drake_graph`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Recopute the config list early and often to have the
# most current information. Do not modify the config list by hand.
config <- drake_config(my_plan)
outdated(config = config) # Which targets are out of date?
config <- make(my_plan) # Run the projects, build the targets.
# Now, everything should be up to date (no targets listed).
outdated(config = config)
# outdated() is sensitive to triggers.
# See the "debug" vignette for more on triggers.
config$trigger <- "always"
outdated(config = config)
})

## End(Not run)
```

`output_sink_hook` *An example* hook *argument to* `make()` *that redirects output messages to files.*

Description

Most users do not need to micromanage hooks.

Usage

```
output_sink_hook(code)
```

Arguments

`code`	code to run to build the target.

Value

A function that you can supply to the hook argument of `make()`.

See Also

`make`, `silencer_hook`, `message_sink_hook`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# Test out the output sink hook on its own.
try(
  output_sink_hook({
    cat(1234)
    stop(5678)
  }),
  silent = FALSE
)
# Create a new workflow plan.
x <- drake_plan(loud = cat(1234), bad = stop(5678))
# Run the project. Standard output (via cat() and print())
# should be suppressed, but messages should persist.
try(make(x, hook = output_sink_hook), silent = FALSE)
})

## End(Not run)
```

`parallelism_choices` *List the types of supported parallel computing in drake.*

Description

These are the possible values of the `parallelism` argument to `make()`.

Usage

```
parallelism_choices(distributed_only = FALSE)
```

Arguments

`distributed_only`
: logical, whether to return only the distributed backend types, such as `Makefile` and `parLapply`

Details

Run `make(..., parallelism = x, jobs = n)` for any of the following values of `x` to distribute targets over parallel units of execution.

'parLapply' launches multiple processes in a single R session using `parallel::parLapply()`. This is single-node, (potentially) multicore computing. It requires more overhead than the `'mclapply'` option, but it works on Windows. If `jobs` is `1` in `make()`, then no 'cluster' is created and no parallelism is used.

'mclapply' uses multiple processes in a single R session. This is single-node, (potentially) multicore computing. Does not work on Windows for `jobs > 1` because `mclapply()` is based on forking.

'future_lapply' opens up a whole trove of parallel backends powered by the `future` and `future.batchtools` packages. First, set the parallel backend globally using `future::plan()`. Then, apply the backend to your drake_plan using `make(..., parallelism = "future_lapply", jobs = ...)`. But be warned: the environment for each target needs to be set up from scratch, so this backend type is higher overhead than either `mclapply` or `parLapply`. Also, the `jobs` argument only applies to the imports. To set the max number of jobs, set the `workers` argument where it exists. For example, call `future::plan(multisession(workers = 4))`, then call `make(your_plan, parallelism = "future_lapply")`. You might also try options(mc.cores = jobs), or see `?future::future::.options` for environment variables that set the max number of jobs.

'Makefile' uses multiple R sessions by creating and running a Makefile. For distributed computing on a cluster or supercomputer, try `make(..., parallelism = 'Makefile', prepend = 'SHELL=./shell.sh')`. You need an auxiliary `shell.sh` file for this, and `shell_file()` writes an example.

Here, Makefile-level parallelism is only used for targets in your workflow plan data frame, not imports. To process imported objects and files, drake selects the best parallel backend for your system and uses the number of jobs you give to the `jobs` argument to `make()`. To use at most 2 jobs for imports and at most 4 jobs for targets, run `make(..., parallelism = 'Makefile', jobs = 2, args = '--j`

Caution: the Makefile generated by `make(..., parallelism = 'Makefile')` is NOT standalone. DO NOT run it outside of `make()` or `make()`. Also, Windows users will need to download and install Rtools.

Value

Character vector listing the types of parallel computing supported.

See Also

`make`, `shell_file`

Examples

```
# See all the parallel computing options.
parallelism_choices()
# See just the distributed computing options.
parallelism_choices(distributed_only = TRUE)
```

`parallel_stages` *Show how* `make()` *will build your targets in successive parallelizable stages.*

Description

The stages determine the order in which `make()` builds the targets.

Usage

```
parallel_stages(config, from_scratch = FALSE)
```

Arguments

config	An configuration list output by make() or drake_config().
from_scratch	logical, whether to assume that the next make() will run from scratch so that all targets are attempted.

Details

Usually, make() divides the targets and imports into parallelizable stages strictly according to the columns in vis_drake_graph(). However, if some targets are out of date, drake looks ahead in the graph until it finds outdated targets for the current stage. The parallel_stages() function takes this behavior into account when it reports a data frame of information on how targets and imports will be divided into parallel stages during the next make().

Value

A data frame of information spelling out how targets are divided into parallelizable stages (according to the stage column).

See Also

next_stage, make, make_with_config

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
config <- drake_config(my_plan) # Get a configuration list.
# Parallel stages for the next make().
parallel_stages(config = config)
# Check the graph to see that the information agrees.
vis_drake_graph(config = config)
# Build the project.
config <- make_with_config(config) # or make(my_plan)
# Nothing to build in the next make().
parallel_stages(config = config)
# Change a dependency and notice how the stages change.
reg2 = function(d){
  d$x3 = d$x^3
  lm(y ~ x3, data = d)
}
parallel_stages(config = config)
})

## End(Not run)
```

plan_analyses *Generate a workflow plan data frame to analyze multiple datasets using multiple methods of analysis.*

Description

Uses wildcards to create a new workflow plan data frame from a template data frame.

Usage

```
plan_analyses(plan, datasets)
```

Arguments

plan	workflow plan data frame of analysis methods. The commands in the command column must have the dataset__ wildcard where the datasets go. For example, one command could be lm(dataset__). Then, the commands in the output will include lm(your_dataset_1), lm(your_dataset_2), etc.
datasets	workflow plan data frame with instructions to make the datasets.

Value

An evaluated workflow plan data frame of analysis targets.

See Also

plan_summaries, make, drake_plan

Examples

```
# Create the piece of the workflow plan for the datasets.
datasets <- drake_plan(
  small = simulate(5),
  large = simulate(50))
# Create a template for the analysis methods.
methods <- drake_plan(
  regression1 = reg1(dataset__),
  regression2 = reg2(dataset__))
# Evaluate the wildcards to create the part of the workflow plan
# encoding the analyses of the datasets.
ans <- plan_analyses(methods, datasets = datasets)
ans
# For the final workflow plan, row bind the pieces together.
my_plan <- rbind(datasets, ans)
my_plan
```

plan_summaries *Generate a workflow plan data frame for summarizing multiple analyses of multiple datasets multiple ways.*

Description

Uses wildcards to create a new workflow plan data frame from a template data frame.

Usage

```
plan_summaries(plan, analyses, datasets, gather = rep("list", nrow(plan)))
```

Arguments

plan	workflow plan data frame with commands for the summaries. Use the analysis__ and dataset__ wildcards just like the dataset__ wildcard in analyses().
analyses	workflow plan data frame of analysis instructions
datasets	workflow plan data frame with instructions to make or import the datasets.
gather	Character vector, names of functions to gather the summaries. If not NULL, the length must be the number of rows in the plan. See the gather() function for more.

Value

An evaluated workflow plan data frame of instructions for computing summaries of analyses and datasets. analyses of multiple datasets in multiple ways.

See Also

plan_analyses, make, drake_plan

Examples

```
# Create the part of the workflow plan data frame for the datasets.
datasets <- drake_plan(
  small = simulate(5),
  large = simulate(50))
# Create a template workflow plan containing the analysis methods.
methods <- drake_plan(
  regression1 = reg1(dataset__),
  regression2 = reg2(dataset__))
# Generate the part of the workflow plan to analyze the datasets.
analyses <- plan_analyses(methods, datasets = datasets)
# Create a template workflow plan dataset with the
# types of summaries you want.
summary_types <- drake_plan(
  summ = summary(analysis__),
  coef = coefficients(analysis__))
```

```
# Evaluate the appropriate wildcards to encode the summary targets.
plan_summaries(summary_types, analyses, datasets, gather = NULL)
plan_summaries(summary_types, analyses, datasets)
plan_summaries(summary_types, analyses, datasets, gather = "list")
summs <- plan_summaries(
  summary_types, analyses, datasets, gather = c("list", "rbind"))
# For the final workflow plan, row bind the pieces together.
my_plan <- rbind(datasets, analyses, summs)
my_plan
```

`predict_runtime` *Predict the elapsed runtime of the next call to 'make()'.*

Description

This function simply sums the elapsed build times from `rate_limiting_times()`, and this feature is experimental. To date, the accuracy/precision of the results has not yet been confirmed by performance studies.

Usage

```
predict_runtime(config, targets = NULL, future_jobs = 1,
  from_scratch = FALSE, targets_only = FALSE, digits = 3)
```

Arguments

`config`	option internal runtime parameter list of `make(...)`, produced by both `make()` and `drake_config()`.
`targets`	Character vector, names of targets. Predict the runtime of building these targets plus dependencies. Defaults to all targets.
`future_jobs`	hypothetical number of jobs assumed for the predicted runtime. assuming this number of jobs.
`from_scratch`	logical, whether to predict a `make()` build from scratch or to take into account the fact that some targets may be already up to date and therefore skipped.
`targets_only`	logical, whether to factor in just the targets into the calculations or use the build times for everything, including the imports
`digits`	number of digits for rounding the time

Details

For the results to make sense, the previous build times of all targets need to be available (automatically cached by `make()`). Otherwise, `predict_runtime()` will warn you and tell you which targets have missing times.

Value

A `lubridate` `Duration` object with the predicted runtime of the next `make()`.

See Also

rate_limiting_times, build_times make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
config <- make(my_plan) # Run the project, build the targets.
predict_runtime(config, digits = 4) # Everything is up to date.
predict_runtime(config, digits = 4, from_scratch = TRUE) # 1 job
# Assumes you clean() out your project and start from scratch with 2 jobs.
predict_runtime(config, future_jobs = 2, digits = 4, from_scratch = TRUE)
# Predict the runtime of just building targets
# "small" and "large".
predict_runtime(
  config,
  targets = c("small", "large"),
  from_scratch = TRUE,
  digits = 4
)
})

## End(Not run)
```

---

progress *Get the build progress of your targets during a* make().

Description

Objects that drake imported, built, or attempted to build are listed as "finished" or "in progress". Skipped objects are not listed.

Usage

```
progress(..., list = character(0), no_imported_objects = FALSE,
  imported_files_only = logical(0), path = getwd(), search = TRUE,
  cache = drake::get_cache(path = path, search = search, verbose = verbose),
  verbose = TRUE, jobs = 1)
```

Arguments

...	objects to load from the cache, as names (unquoted) or character strings (quoted). Similar to ... in remove(...).
list	character vector naming objects to be loaded from the cache. Similar to the list argument of remove().
no_imported_objects	logical, whether to only return information about imported files and targets with commands (i.e. whether to ignore imported objects that are not files).

imported_files_only
: logical, deprecated. Same as no_imported_objects. Use the no_imported_objects argument instead.

path
: Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.

search
: If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.

cache
: optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.

verbose
: whether to print console messages

jobs
: number of jobs/workers for parallel processing

Value

The build progress of each target reached by the current make() so far.

See Also

diagnose, session, built, imported, readd, drake_plan, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
# Watch the changing progress() as make() is running.
progress() # List all the targets reached so far.
progress(small, large) # Just see the progress of some targets.
progress(list = c("small", "large")) # Same as above.
progress(no_imported_objects = TRUE) # Ignore imported R objects.
})

## End(Not run)
```

---

prune_drake_graph *Prune the dependency network of your project.*

Description

igraph objects are used internally to represent the dependency network of your workflow. See config(my_plan)$graph from the basic example.

Usage

```
prune_drake_graph(graph, to = igraph::V(graph)$name, jobs = 1)
```

Arguments

graph	An igraph object to be pruned.
to	Character vector, names of the vertices that draw the line for pruning. The pruning process removes all vertices downstream of to.
jobs	Number of jobs for light parallelism (on non-Windows machines).

Details

For a supplied graph, take the subgraph of all combined incoming paths to the vertices in to. In other words, remove the vertices after to from the graph.

Value

A pruned igraph object representing the dependency network of the workflow.

See Also

build_drake_graph, config, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Build the igraph object representing the workflow dependency network.
# You could also use drake_config(my_plan)$graph
graph <- build_drake_graph(my_plan)
# The default plotting is not the greatest,
# but you will get the idea.
plot(graph)
# Prune the graph: that is, remove the nodes downstream
# from 'small' and 'large'
pruned <- prune_drake_graph(graph = graph, to = c("small", "large"))
plot(pruned)
})

## End(Not run)
```

---

rate_limiting_times *Return a data frame of elapsed build times of the rate-limiting targets of a drake project.*

Description

This function produces a conservative estimate for predict_runtime() for when parallel computing is used in make(). This feature is experimental. The accuracy, precision, and utility of these supposedly rate-limiting times has not been confirmed by rigorous performance studies.

Usage

```
rate_limiting_times(config, targets = NULL, from_scratch = FALSE,
  targets_only = FALSE, future_jobs = 1, digits = 3)
```

Arguments

config	option internal runtime parameter list of make(...), produced by both make() and drake_config().
targets	Character vector, names of targets. Find the rate-limiting times for building these targets plus dependencies. Defaults to all targets.
from_scratch	logical, whether to assume next hypothetical call to make() is a build from scratch (after clean()).
targets_only	logical, whether to factor in just the targets or use times from everything, including the imports.
future_jobs	hypothetical number of jobs assumed for the predicted runtime. assuming this number of jobs.
digits	number of digits for rounding the times.

Details

The stage column of the returned data frame is an index that denotes a parallelizable stage. Within each stage during make(), the targets are divided among the available jobs. For rate_limiting_times(), we assume the targets are divided evenly among the jobs and one job gets all the slowest targets. The build times of this hypothetical pessimistic job are returned for each stage.

By default from_scratch is FALSE. That way, rate_limiting_times() takes into account that some targets are already up to date, meaning their elapsed build times will be instant during the next make().

For the results to make sense, the previous build times of all targets need to be available (automatically cached by make()). Otherwise, rate_limiting_times() will warn you and tell you which targets have missing times.

Value

A data frame of times of the worst-case scenario rate-limiting targets in each parallelizable stage.

See Also

predict_runtime, build_times make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
config <- make(my_plan) # Run the project, build the targets.
rate_limiting_times(config) # Everything is up to date.
```

```
# Assume everything runs from scratch with 1 job.
rate_limiting_times(config, from_scratch  = TRUE, digits = 4)
# With 2 jobs, some of the targets are not rate-limiting.
rate_limiting_times(
  config,
  future_jobs = 2,
  from_scratch = TRUE,
  digits = 4
)
# Find the rate-limiting times of just building targets
# "small" and "large".
rate_limiting_times(
  config,
  targets = c("small", "large"),
  from_scratch = TRUE,
  digits = 4
)
})

## End(Not run)
```

---

readd *Read and return a drake target or import from the cache.*

Description

Does not delete the item from the cache.

Usage

```
readd(target, character_only = FALSE, path = getwd(), search = TRUE,
  cache = drake::get_cache(path = path, search = search, verbose = verbose),
  namespace = NULL, verbose = TRUE)
```

Arguments

target	If character_only is TRUE, target is a character string naming the object to read. Otherwise, target is an unquoted symbol with the name of the object. Note: target could be the name of an imported object.
character_only	logical, whether name should be treated as a character or a symbol (just like character.only in library()).
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.

namespace	character scalar, name of an optional storr namespace to read from.
verbose	whether to print console messages

Value

The cached value of the target.

See Also

loadd, cached, built, link{imported}, drake_plan, make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
readd(reg1) # Return imported object 'reg1' from the cache.
readd(small) # Return targets 'small' from the cache.
readd("large", character_only = TRUE) # Return 'large' from the cache.
# For external files, only the fingerprint/hash is stored.
readd("'report.md'")
})

## End(Not run)
```

---

read_drake_config *Read the cached* drake_config() *list from the last* make().

Description

See drake_config() for more information about drake's internal runtime configuration parameter list.

Usage

```
read_drake_config(path = getwd(), search = TRUE, cache = NULL,
  verbose = 1, jobs = 1, envir = parent.frame())
```

Arguments

path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.
verbose	whether to print console messages

jobs	number of jobs for light parallelism. Supports 1 job only on Windows.
envir	Optional environment to fill in if config$envir was not cached. Defaults to your workspace.

Value

The cached master internal configuration list of the last make().

See Also

make

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
# Retrieve the master internal configuration list from the cache.
read_drake_config()
})

## End(Not run)
```

---

read_drake_graph *Read the igraph dependency network from your last attempted call to make().*

Description

For more user-friendly graphing utilities, see vis_drake_graph() and related functions.

Usage

```
read_drake_graph(path = getwd(), search = TRUE, cache = NULL,
  verbose = 1, ...)
```

Arguments

path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.
verbose	logical, whether to print console messages
...	arguments to visNetwork() via vis_drake_graph()

Value

An igraph object representing the dependency network of the workflow.

See Also

vis_drake_graph, read_drake_config

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
# Retrieve the igraph network from the cache.
g <- read_drake_graph()
class(g) # "igraph"
})

## End(Not run)
```

---

read_drake_meta *Read the metadata of a target or import.*

Description

The metadata helps determine if the target is up to date or outdated. The metadata of imports is used to compute the metadata of targets.

Usage

```
read_drake_meta(targets = NULL, path = getwd(), search = TRUE,
  cache = NULL, verbose = 1, jobs = 1)
```

Arguments

targets	character vector, names of the targets to get metadata. If NULL, all metadata is collected.
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
cache	optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.
verbose	whether to print console messages
jobs	number of jobs for light parallelism. Supports 1 job only on Windows.

Details

Target metadata is computed with `drake_meta()` and then `drake:::finish_meta()`. This metadata corresponds to the state of the target immediately after it was built or imported in the last `make()` that did not skip it. The exception to this is the `$missing` element of the metadata, which indicates if the target/import was missing just *before* it was built.

Value

The cached master internal configuration list of the last `make()`.

See Also

`dependency_profile`, `make`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
# Retrieve the build decision metadata for one target.
read_drake_meta(targets = "small")
# Retrieve the build decision metadata for all targets,
# parallelizing over 2 jobs.
read_drake_meta(jobs = 2)
})

## End(Not run)
```

---

read_drake_plan *Read the workflow plan from your last attempted call to* `make()`.

Description

Uses the cache.

Usage

```
read_drake_plan(path = getwd(), search = TRUE, cache = NULL,
  verbose = TRUE)
```

Arguments

`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project.
`search`	logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.

cache	optional drake cache. See codenew_cache(). If cache is supplied, the path and search arguments are ignored.
verbose	whether to print console messages

Value

A workflow plan data frame.

See Also

read_drake_config

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
read_drake_plan() # Retrieve the workflow plan data frame from the cache.
})

## End(Not run)
```

---

recover_cache *Load an existing drake files system cache if it exists or create a new one otherwise.*

Description

Does not work with in-memory caches such as storr_environment().

Usage

```
recover_cache(path = drake::default_cache_path(),
  short_hash_algo = drake::default_short_hash_algo(),
  long_hash_algo = drake::default_long_hash_algo(), force = FALSE,
  verbose = TRUE, fetch_cache = NULL)
```

Arguments

path	file path of the cache
short_hash_algo	short hash algorithm for the cache. See default_short_hash_algo() and make()
long_hash_algo	long hash algorithm for the cache. See default_long_hash_algo() and make()
force	logical, whether to load the cache despite any back compatibility issues with the running version of drake.

verbose	logical, whether to print the file path of the cache.
fetch_cache	character vector containing lines of code. The purpose of this code is to fetch the storr cache with a command like storr_rds() or storr_dbi(), but customized.

Value

A drake/storr cache.

See Also

new_cache, this_cache, get_cache

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
clean(destroy = TRUE)
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build all the targets.
x <- recover_cache(".drake") # Recover the project's storr cache.
})

## End(Not run)
```

---

render_drake_graph *Render a visualization using the data frames generated by* dataframes_graph().

Description

This function is called inside vis_drake_graph(), which typical users call more often.

Usage

```
render_drake_graph(graph_dataframes, file = character(0),
  layout = "layout_with_sugiyama", direction = "LR", hover = TRUE,
  main = graph_dataframes$default_title, selfcontained = FALSE,
  navigationButtons = TRUE, ncol_legend = 1, ...)
```

Arguments

graph_dataframes	list of data frames generated by dataframes_graph(). There should be 3 data frames: nodes, edges, and legend_nodes.
file	Name of HTML file to save the graph. If NULL or character(0), no file is saved and the graph is rendered and displayed within R.

layout	name of an igraph layout to use, such as 'layout_with_sugiyama' or 'layout_as_tree'. Be careful with 'layout_as_tree': the graph is a directed acyclic graph, but not necessarily a tree.
direction	an argument to visNetwork::visHierarchicalLayout() indicating the direction of the graph. Options include 'LR', 'RL', 'DU', and 'UD'. At the time of writing this, the letters must be capitalized, but this may not always be the case ;) in the future.
hover	logical, whether to show the command that generated the target when you hover over a node with the mouse. For imports, the label does not change with hovering.
main	title of the graph
selfcontained	logical, whether to save the file as a self-contained HTML file (with external resources base64 encoded) or a file with external resources placed in an adjacent directory. If TRUE, pandoc is required.
navigationButtons	logical, whether to add navigation buttons with visNetwork::visInteraction(navigationButtons =
ncol_legend	number of columns in the legend nodes
...	arguments passed to visNetwork().

Value

A visNetwork graph.

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Instead of jumpting right to vis_drake_graph(), get the data frames
# of nodes, edges, and legend nodes.
config <- drake_config(my_plan) # Internal configuration list
graph <- dataframes_graph(config)
# You can pass the data frames right to render_drake_graph()
# (as in vis_drake_graph()) or you can create
# your own custom visNewtork graph.
render_drake_graph(graph, width = '100%') # Width is passed to visNetwork.
})

## End(Not run)
```

---

rescue_cache *Try to repair a drake cache that is prone to throwing* storr*-related errors.*

Description

Sometimes, storr caches may have dangling orphaned files that prevent you from loading or cleaning. This function tries to remove those files so you can use the cache normally again.

Usage

```
rescue_cache(targets = NULL, path = getwd(), search = TRUE, verbose = 1,
  force = FALSE, cache = drake::get_cache(path = path, search = search,
  verbose = verbose, force = force), jobs = 1, garbage_collection = FALSE)
```

Arguments

targets	Character vector, names of the targets to rescue. As with many other drake utility functions, the word target is defined generally in this case, encompassing imports as well as true targets. If targets is NULL, everything in the cache is rescued.
path	same as for get_cache()
search	same as for get_cache()
verbose	same as for get_cache()
force	same as for get_cache()
cache	a 'storr' cache object
jobs	number of jobs for light parallelism (disabled on Windows)
garbage_collection	logical, whether to do garbage collection as a final step. See drake_gc and clean for details.

Value

The rescued drake/storr cache.

See Also

get_cache, cached, drake_gc, clean

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build targets. This creates the cache.
# Remove dangling cache files that could cause errors.
rescue_cache(jobs = 2)
# Alternatively, just rescue targets 'small' and 'large'.
# Rescuing specific targets is usually faster.
rescue_cache(targets = c("small", "large"))
})

## End(Not run)
```

`r_recipe_wildcard` *Show the R recipe wildcard for* `make(..., parallelism = "Makefile")`.

Description

Relevant to `"Makefile"` parallelism only.

Usage

```
r_recipe_wildcard()
```

Value

The R recipe wildcard.

See Also

`default_recipe_command`

Examples

```
r_recipe_wildcard()
```

`shell_file` *Write an example* `shell.sh` *file required by* `make(..., parallelism = 'Makefile', prepend = 'SHELL=./shell.sh')`.

Description

This function also does a ‘chmod +x‘ to enable execute permissions.

Usage

```
shell_file(path = "shell.sh", overwrite = FALSE)
```

Arguments

`path`	file path of the shell file
`overwrite`	logical, whether to overwrite a possible destination file with the same name

Value

The return value of the call to `file.copy()` that wrote the shell file.

See Also

make, max_useful_jobs, parallelism_choices, drake_batchtools_tmpl_file, drake_example, drake_examples

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# Write shell.sh to your working directory.
# Read the parallelism vignette to learn how it is used
# in Makefile parallelism.
shell_file()
})

## End(Not run)
```

---

short_hash *Get the short hash algorithm of a drake cache.*

Description

See the advanced storage tutorial at https://ropensci.github.io/drake/articles/storage.html for details.

Usage

```
short_hash(cache = drake::get_cache(verbose = verbose), verbose = verbose)
```

Arguments

cache	drake cache
verbose	whether to print console messages

Value

A character vector naming a hash algorithm.

See Also

default_short_hash_algo, default_long_hash_algo

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Get the code with drake_example("basic").
# Run the project and return the internal master configuration list.
config <- make(my_plan)
# Locate the storr cache.
cache <- config$cache
# Get the short hash algorithm of the cache.
short_hash(cache)
})

## End(Not run)
```

`silencer_hook` *An example* hook *argument to* `make()` *that redirects output and error messages*

Description

Most users do not need to micromanage hooks.

Usage

```
silencer_hook(code)
```

Arguments

code	code to run to build the target.

Value

A function that you can supply to the hook argument of `make()`.

See Also

`make`, `message_sink_hook`, `output_sink_hook`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
# Test out the silencer hook on its own.
try(
  silencer_hook({
    cat(1234)
    stop(5678)
  }),
  silent = FALSE
```

```
)
# Make a new workflow plan.
x <- drake_plan(loud = cat(1234), bad = stop(5678))
# Test out the silencer hook on a drake project.
# All output should be suppressed.
try(make(x, hook = silencer_hook), silent = FALSE)
})

## End(Not run)
```

`target_namespaces` *For drake caches, list the* `storr` *cache namespaces that store target-level information.*

Description

Ordinary users do not need to worry about this function. It is just another window into `drake`'s internals.

Usage

```
target_namespaces(default = storr::storr_environment()$default_namespace)
```

Arguments

`default`	name of the default `storr` namespace

Value

A character vector of `storr` namespaces that store target-level information.

See Also

`make`

Examples

```
target_namespaces()
```

this_cache *Get the cache at the exact file path specified.*

Description

This function does not apply to in-memory caches such as `storr_environment()`.

Usage

```
this_cache(path = drake::default_cache_path(), force = FALSE,
  verbose = TRUE, fetch_cache = NULL)
```

Arguments

`path`	file path of the cache
`force`	logical, whether to load the cache despite any back compatibility issues with the running version of drake.
`verbose,`	whether to print the file path of the cache.
`fetch_cache`	character vector containing lines of code. The purpose of this code is to fetch the `storr` cache with a command like `storr_rds()` or `storr_dbi()`, but customized. This feature is experimental.

Value

A drake/storr cache at the specified path, if it exists.

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
clean(destroy = TRUE)
try(x <- this_cache(), silent = FALSE) # The cache does not exist yet.
load_basic_example() # Get the code with drake_example("basic").
make(my_plan) # Run the project, build the targets.
y <- this_cache() # Now, there is a cache.
z <- this_cache(".drake") # Same as above.
manual <- new_cache("manual_cache") # Make a new cache.
manual2 <- get_cache("manual_cache") # Get the new cache.
})

## End(Not run)
```

tracked *List the targets and imports that are reproducibly tracked.*

Description

In other words, list all the nodes in your project's dependency network.

Usage

```
tracked(plan = drake_plan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), jobs = 1, verbose = TRUE)
```

Arguments

plan	workflow plan data frame, same as for function make().
targets	names of targets to build, same as for function make().
envir	environment to import from, same as for function make().
jobs	number of jobs to accelerate the construction of the dependency graph. A light mclapply-based parallelism is used if your operating system is not Windows.
verbose	logical, whether to print progress messages to the console.

Value

A character vector with the names of reproducibly-tracked targets.

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Load the canonical example for drake.
# List all the targets/imports that are reproducibly tracked.
tracked(my_plan)
})

## End(Not run)
```

triggers *List the available drake triggers.*

Description

Triggers are target-level rules that tell make() how to know if a target is outdated or up to date.

Usage

```
triggers()
```

Details

By default, `make()` builds targets that need updating and skips over the ones that are already up to date. In other words, a change in a dependency, workflow plan command, or file, or the lack of the target itself, *triggers* the build process for the target. You can relax this behavior by choosing a trigger for each target. Set the trigger for each target with a `"trigger"` column in your workflow plan data frame. The `triggers()` function lists the available triggers:

- 'any': Build the target if any of the other triggers activate (default).
- 'command': Build if the workflow plan command has changed since last time the target was built. Also built if `missing` is triggered.
- 'depends': Build if any of the target's dependencies has changed since the last `make()`. Also build if `missing` is triggered.
- 'file': Build if the target is a file and that output file is either missing or corrupted. Also build if `missing` is triggered.
- 'missing': Build if the target itself is missing. Always applies.

Value

A character vector with the names of the available triggers.

See Also

`drake_plan`, `make`

Examples

```
triggers()
## Not run:
test_with_dir("Quarantine side effects.", {
load_basic_example() # Load drake's canonical example.
my_plan[["trigger"]] <- "command"
# You can have different triggers for different targets.
my_plan[["trigger"]][1] <- "file"
make(my_plan) # Run the project, build the targets.
# Change an imported dependency function.
reg2 <- function(d) {
  d$x3 <- d$x ^ 3
  lm(y ~ x3, data = d)
}
# Nothing changes! To react to `reg2`, you would need the
# "any" or "depends" trigger.
make(my_plan)
# You can use a global trigger if your workflow plan
# does not have a 'trigger' column.
my_plan[["trigger"]] <- NULL # Would override the global trigger.
make(my_plan, trigger = "missing") # Just build missing targets.
})

## End(Not run)
```

vis_drake_graph *Show an interactive visual network representation of your drake project.*

Description

To save time for repeated plotting, this function is divided into `dataframes_graph()` and `render_drake_graph()`.

Usage

```
vis_drake_graph(config, file = character(0), selfcontained = FALSE,
  build_times = TRUE, digits = 3, targets_only = FALSE,
  split_columns = FALSE, font_size = 20, layout = "layout_with_sugiyama",
  main = NULL, direction = "LR", hover = TRUE, navigationButtons = TRUE,
  from = NULL, mode = c("out", "in", "all"), order = NULL,
  subset = NULL, ncol_legend = 1, make_imports = TRUE,
  from_scratch = FALSE, ...)
```

Arguments

`config`	Master configuration list produced by both `make()` and `drake_config()`.
`file`	Name of HTML file to save the graph. If `NULL` or `character(0)`, no file is saved and the graph is rendered and displayed within R.
`selfcontained`	logical, whether to save the `file` as a self-contained HTML file (with external resources base64 encoded) or a file with external resources placed in an adjacent directory. If `TRUE`, pandoc is required.
`build_times`	logical, whether to print the `build_times()` in the graph.
`digits`	number of digits for rounding the build times
`targets_only`	logical, whether to skip the imports and only show the targets in the workflow plan.
`split_columns`	logical, whether to break up the columns of nodes to make the aspect ratio of the rendered graph closer to 1:1. This improves the viewing experience, but the columns no longer strictly represent parallelizable stages of build items. (Although the targets/imports in each column are still conditionally independent, there may be more conditional independence than the graph indicates.)
`font_size`	numeric, font size of the node labels in the graph
`layout`	name of an igraph layout to use, such as 'layout_with_sugiyama' or 'layout_as_tree'. Be careful with 'layout_as_tree': the graph is a directed acyclic graph, but not necessarily a tree.
`main`	title of the graph
`direction`	an argument to `visNetwork::visHierarchicalLayout()` indicating the direction of the graph. Options include 'LR', 'RL', 'DU', and 'UD'. At the time of writing this, the letters must be capitalized, but this may not always be the case ;) in the future.

hover	logical, whether to show the command that generated the target when you hover over a node with the mouse. For imports, the label does not change with hovering.
navigationButtons	logical, whether to add navigation buttons with visNetwork::visInteraction(navigationButtons =
from	Optional character vector of target/import names. If from is nonempty, the graph will restrict itself to a neighborhood of from. Control the neighborhood with mode and order.
mode	Which direction to branch out in the graph to create a neighborhood around from. Use "in" to go upstream, "out" to go downstream, and "all" to go both ways and disregard edge direction altogether.
order	How far to branch out to create a neighborhood around from (measured in the number of nodes). Defaults to as far as possible.
subset	Optional character vector of of target/import names. Subset of nodes to display in the graph. Applied after from, mode, and order. Be advised: edges are only kept for adjacent nodes in subset. If you do not select all the intermediate nodes, edges will drop from the graph.
ncol_legend	number of columns in the legend nodes
make_imports	logical, whether to import external files and objects from the user's workspace to determine which targets are up to date. If FALSE, the computation is faster, but all the relevant information is drawn from the cache and may be out of date.
from_scratch	logical, whether to assume all the targets will be made from scratch on the next make(). Makes all targets outdated, but keeps information about build progress in previous make()s.
...	other arguments passed to visNetwork::visNetwork() to plot the graph.

Value

A visNetwork graph.

See Also

build_drake_graph

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
config <- load_basic_example() # Get the code with drake_example("basic").
# Plot the network graph representation of the workflow.
vis_drake_graph(config, width = '100%') # The width is passed to visNetwork
config <- make(my_plan) # Run the project, build the targets.
vis_drake_graph(config) # The red nodes from before are now green.
# Plot a subgraph of the workflow.
vis_drake_graph(
  config,
  from = c("small", "reg2"),
```

```
  to = "summ_regression2_small"
)
})

## End(Not run)
```

Index