Package 'drake'

November 5, 2017

Title Data Frames in R for Make

Version 4.4.0

Description A solution for reproducible code and high-performance computing.

License GPL-3

Depends R (>= 3.2.0)

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

Suggests abind, future.batchtools, MASS, methods, rmarkdown, tibble

VignetteBuilder knitr

URL https://github.com/wlandau-lilly/drake

BugReports https://github.com/wlandau-lilly/drake/issues

RoxygenNote 6.0.1

NeedsCompilation no

Author William Michael Landau [aut, cre], Alex Axthelm [ctb], Jasper Clarkberg [ctb], Eli Lilly and Company [cph]

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

Repository CRAN

Date/Publication 2017-11-05 06:02:56 UTC

R topics documented:

`drake-package` *Drake is a workplan manager and build system, a scalable solution for reproducibility and high-performance computing.*

Description

Drake is a workplan manager and build system, a scalable solution for reproducibility and high-performance computing.

Author(s)

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

References

https://github.com/wlandau-lilly/drake

Examples

```
## Not run:
library(drake)
load_basic_example()
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.
readd(small) # Read/load from the cache.
loadd(large)
head(large)
clean() # Restart from scratch
make(my_plan, jobs = 2) # Distribute over 2 parallel jobs.
clean()
make(my_plan, jobs = 4, parallelism = "Makefile") # 4 parallel R sessions.
make(my_plan, jobs = 4, parallelism = "Makefile") # Everything up to date.
clean(destroy = TRUE) # Totally remove the cache.
unlink(c("Makefile", "report.Rmd"))

## End(Not run)
```

---

analyses *Function* analyses

Description

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

Usage

```
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 instructions

See Also

summaries, make, workplan

Examples

```
datasets <- workplan(
  small = simulate(5),
  large = simulate(50))
methods <- workplan(
  regression1 = reg1(..dataset..),
  regression2 = reg2(..dataset..))
analyses(methods, datasets = datasets)
```

---

as_file *Function* as_file

Description

Converts an ordinary character string into a filename understandable by drake. In other words, as_file(x) just wraps single quotes around x.

Usage

```
as_file(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

```
as_file("my_file.rds")
```

available_hash_algos *Function available_hash_algos*

Description

List the available hash algorithms.

Usage

```
available_hash_algos()
```

Examples

```
available_hash_algos()
```

backend *Function* backend

Description

Select the specific `future`-style parallel backend for `make(..., parallelism = "future_lapply")`. For more information, please read the documentation, tutorials, and vignettes of the R packages `future` and `future.batchtools`.

Usage

```
backend(...)
```

Arguments

`...`	arguments to `future::plan()`.

Details

The `backend()` function is exactly the same as `future::plan()`. We provide it only because `workplan()` conflicts with `future::plan()`.

See Also

`workplan`, `make`,

Examples

```
## Not run:
load_basic_example()
library(future) # Use workplan() instead of plan()
backend(multicore) # Same as future::plan(multicore)
make(my_plan, parallelism = "future_lapply")
clean() # Erase the targets to start from scratch.
backend(multisession) # Use separate background R sessions.
make(my_plan, parallelism = "future_lapply")
clean()
library(future.batchtools) # More heavy-duty future-style parallel backends
# https://github.com/HenrikBengtsson/future.batchtools#choosing-batchtools-backend # nolint
backend(batchtools_local)
make(my_plan, parallelism = "future_lapply")
clean()

## End(Not run)
```

---

build *Internal function build*

Description

Function to build a target. For internal use only. the only reason this function is exported is to set up PSOCK clusters efficiently.

Usage

```
build(target, hash_list, config)
```

Arguments

target	name of the target
hash_list	list of hashes that tell which targets are up to date
config	internal configuration list

---

build_graph *Function* build_graph

Description

Make a graph of the dependency structure of your workplan.

Usage

```
build_graph(plan = workplan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, 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.

Details

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

See Also

plot_graph

Examples

```
## Not run:
load_basic_example()
g <- build_graph(my_plan)
class(g)

## End(Not run)
```

---

build_times *Function* build_times

Description

List all the build times. This does 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)
```

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

Value

data.frame of times from system.time

See Also

built

Examples

```
## Not run:
load_basic_example()
make(my_plan)
build_times()

## End(Not run)
```

---

built *Function* built

Description

List all the built (non-imported) objects in the drake cache.

Usage

```
built(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	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

Value

list of imported objects in the cache

See Also

cached, loadd, link{imported}

Examples

```
## Not run:
load_basic_example()
make(my_plan)
built()

## End(Not run)
```

---

cached *Function* cached

Description

Check whether targets are in the cache. If no targets are specified with ... or list, then cached() lists all the items in the drake cache. Read/load a cached item with readd() or loadd().

Usage

```
cached(..., list = character(0), no_imported_objects = FALSE,
  path = getwd(), search = TRUE, cache = drake::get_cache(path = path,
  search = search, verbose = verbose), verbose = TRUE)
```

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

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, workplan, make

Examples

```
## Not run:
load_basic_example()
make(my_plan)
cached(list = 'reg1')
cached(no_imported_objects = TRUE)
cached()

## End(Not run)
```

---

cache_path *Function cache_path*

Description

Returns the file path where the cache is stored. Currently only works with storr file system caches.

Usage

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

Arguments

cache	the cache whose file path you want to know

cache_types *Function cache_types*

Description

List the supported drake cache types.

Usage

```
cache_types()
```

See Also

in_memory_cache_types, new_cache, get_cache, default_cache_type

Examples

```
cache_types()
```

check *Function* check

Description

Check a workflow plan, etc. for obvious errors such as circular dependencies and missing input files.

Usage

```
check(plan = workplan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), cache = drake::get_cache(verbose = verbose),
  verbose = TRUE)
```

Arguments

plan	workflow plan data frame, possibly from workplan().
targets	character vector of targets to make
envir	environment containing user-defined functions
cache	optional drake cache. See new_cache()
verbose	logical, whether to log progress to the console.

Value

invisibly return plan

See Also

ink{workplan}, make

Examples

```
## Not run:
load_basic_example()
check(my_plan)
unlink('report.Rmd')
check(my_plan)

## End(Not run)
```

---

clean *Function* clean

Description

Cleans up all work done by make().

Usage

```
clean(..., list = character(0), destroy = FALSE, path = getwd(),
  search = TRUE, cache = NULL, verbose = TRUE)
```

Arguments

...	targets to remove from the cache, as names (unquoted) or character strings (quoted). Similar to ... in remove(...).
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

Details

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.

See Also

`prune`, `make`,

Examples

```
## Not run:
load_basic_example()
make(my_plan)
cached(no_imported_objects = TRUE)
clean(summ_regression1_large, small)
cached(no_imported_objects = TRUE)
make(my_plan)
clean()
clean(destroy = TRUE)

## End(Not run)
```

`config` *Function config*

Description

Compute the internal runtime parameter list of `make()`. This could save time if you are planning multiple function calls of functions like `outdated()` or `plot_graph()`. Drake needs to import and cache files and objects to compute the configuration list, which in turn supports user-side functions to help with visualization and parallelism. The result differs from `make(..., imports_only = TRUE)` in that the graph includes both the targets and the imports, not just the imports.

Usage

```
config(plan = workplan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, hook = function(code) {
  force(code) }, cache = drake::get_cache(verbose = verbose),
  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, clear_progress = FALSE,
  graph = 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
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
clear_progress	logical, whether to clear the cached progress of the targets readable by
graph	igraph object representing the workflow plan network progress()

See Also

workplan, make, plot_graph

Examples

```
## Not run:
load_basic_example()
con <- config(my_plan)
outdated(my_plan, config = con)
missed(my_plan, config = con)
max_useful_jobs(my_plan, config = con)
plot_graph(my_plan, config = con)
dataframes_graph(my_plan, config = con)

## End(Not run)
```

configure_cache *Function configure_cache*

Description

configure a cache for drake. This 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),
  clear_progress = FALSE, overwrite_hash_algos = FALSE, verbose = TRUE)
```

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.
clear_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

See Also

`default_short_hash_algo`, `default_long_hash_algo`

Examples

```
## Not run:
load_basic_example()
config <- make(my_plan)
cache <- config$cache
long_hash(cache)
cache <- configure_cache(
  cache = cache,
  long_hash_algo = "murmur32",
  overwrite_hash_algos = TRUE
```

```
)
long_hash(cache) # long hash algorithm. See ?default_long_hash_algorithm.
make(my_plan) # Changing the long hash puts targets out of date.

## End(Not run)
```

---

dataframes_graph *Function* dataframes_graph

Description

Get the information about nodes, edges, and the legend/key so you can plot your own custom visNetwork.

Usage

```
dataframes_graph(plan = workplan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, hook = function(code) {
  force(code) }, cache = drake::get_cache(verbose = verbose), jobs = 1,
  parallelism = drake::default_parallelism(), packages = rev(.packages()),
  prework = character(0), build_times = TRUE, digits = 3,
  targets_only = FALSE, split_columns = FALSE, font_size = 20,
  config = NULL, from = NULL, mode = c("out", "in", "all"),
  order = NULL, subset = NULL, from_scratch = FALSE)
```

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(). config$envir is ignored in favor of envir.
verbose	logical, whether to output messages to the console.
hook	same as for make
cache	optional drake cache. Only used if the config argument is NULL (default). See codenew_cache().
jobs	The outdated() function is called internally, and it needs to import objects and examine your input files to see what has been updated. This could take some time, and parallel computing may be needed to speed up the process. The jobs argument is number of parallel jobs to use for faster computation.
parallelism	Choice of parallel backend to speed up the computation. Execution order in make() is slightly different when parallelism equals 'Makefile' because in that case, all the imports are imported before any target is built. Thus, the arrangement in the graph is different for Makefile parallelism. See ?parallelism_choices for details.
packages	same as for make

prework	same as for make
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
config	option internal runtime parameter list of make(...), produced with config(). config$envir is ignored. Otherwise, computing this in advance could save time if you plan multiple calls to dataframes_graph(). If not NULL, config overrides all arguments except build_times, digits, targets_only, split_columns, and font_size.
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.
from_scratch	logical, whether to assume that all targets are out of date and the next make() will happen from scratch. Setting to TRUE will prevent the graph from showing you which targets are up to date, but it makes computing the graph much faster.

Value

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

See Also

plot_graph, build_graph

Examples

```
## Not run:
load_basic_example()
```

```
raw_graph <- dataframes_graph(my_plan)
smaller_raw_graph <- dataframes_graph(
  my_plan,
  from = c("small", "reg2"),
  to = "summ_regression2_small"
)
str(raw_graph)
# Plot your own custom visNetwork graph
library(magrittr)
library(visNetwork)
visNetwork(nodes = raw_graph$nodes, edges = raw_graph$edges) %>%
  visLegend(useGroups = FALSE, addNodes = raw_graph$legend_nodes) %>%
  visHierarchicalLayout(direction = 'LR')

## End(Not run)
```

`default_cache_path` *Function default_cache_path*

Description

default drake cache path

Usage

```
default_cache_path()
```

Examples

```
default_cache_path()
```

`default_cache_type` *Function default_cache_type*

Description

Name the default type of drake cache.

Usage

```
default_cache_type()
```

See Also

`new_cache`, `get_cache`, `default_cache_type`

Examples

```
default_cache_type()
```

default_graph_title *Function default_graph_title*

Description

Default title of the graph from plot_graph().

Usage

```
default_graph_title(parallelism = drake::parallelism_choices(distributed_only
  = FALSE), split_columns = FALSE)
```

Arguments

parallelism	Mode of parallelism intended for the workplan. See parallelism_choices().
split_columns	logical, whether the columns were split in dataframes_graph() or plot_graph() with the split_columns argument.

See Also

dataframes_graph, plot_graph

default_long_hash_algo *Default long hash algorithm for* make()

Description

Hashing is advanced. Most users do not need to know about this function.

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.

See Also

`make`, `available_hash_algos`

Examples

```
default_long_hash_algo()
```

`default_Makefile_args` *Function* `default_Makefile_args`

Description

Configures default arguments to `system2()` to run Makefiles.

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` *Function* `default_Makefile_command`

Description

Give the default command argument to `make()`

Usage

```
default_Makefile_command()
```

Examples

```
default_Makefile_command()
```

`default_parallelism` *Function* `default_parallelism`

Description

Default parallelism for `make()`: `'parLapply'` for Windows machines and `'mclapply'` for other platforms.

Usage

```
default_parallelism()
```

Value

default parallelism option for the current platform

See Also

`make`, `shell_file`

Examples

```
default_parallelism()
```

default_recipe_command *default_recipe_command*

Description

Function to give the default recipe command for Makefile parallelism.

Usage

```
default_recipe_command()
```

Details

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

See Also

`Makefile_recipe`

default_short_hash_algo *Default short hash algorithm for* `make()`

Description

Hashing is advanced. Most users do not need to know about this function.

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.

See Also

`make`, `available_hash_algos`

Examples

```
default_short_hash_algo()
```

---

default_system2_args *Deprecated function* default_system2_args

---

Description

Use `default_Makefile_args()` instead.

Usage

```
default_system2_args(jobs, verbose)
```

Arguments

jobs	number of jobs
verbose	logical, whether to be verbose

Value

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

See Also

default_Makefile_args

---

deps *Function deps*

Description

List the dependencies of a function or workflow plan command. Or, 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'`

Usage

```
deps(x)
```

Arguments

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

Value

names of dependencies. Files wrapped in single quotes. The other names listed are functions or generic objects.

Examples

```
f <- function(x, y){
  out <- x + y + g(x)
  saveRDS(out, 'out.rds')
}
deps(f)
my_plan <- workplan(
  x = 1 + some_object,
  my_target = x + readRDS('tracked_input_file.rds'),
  return_value = f(x, y, g(z + w))
)
deps(my_plan$command[1])
deps(my_plan$command[2])
deps(my_plan$command[3])
## Not run:
load_basic_example() # Writes 'report.Rmd'.
deps("'report.Rmd'") # dependencies of future knitted output 'report.md'

## End(Not run)
```

diagnose *Function* diagnose

Description

Get the last stored error of a target that failed to build. This 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

See Also

`failed`, `progress`, `readd`, `workplan`, `make`

Examples

```
## Not run:
diagnose()
f <- function(){
  stop("unusual error")
}
bad_plan <- workplan(my_target = f())
make(bad_plan)
failed() # from the last make() only
diagnose() # from all previous make()'s
error <- diagnose(my_target)
str(error)
```

```
error$calls # View the traceback.

## End(Not run)
```

`do_prework` *Internal function do_prework*

Description

Run the `prework` of a `make()`. For internal use only. the only reason this function is exported is to set up PSOCK clusters efficiently.

Usage

```
do_prework(config, verbose_packages)
```

Arguments

`config`	internal configuration list
`verbose_packages`	logical, whether to print package startup messages

`drake_palette` *Function palette*

Description

show color palette for drake. Used in both the console and `plot_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 `plot_graph()`. So if you want to contribute improvements to the palette, please both `drake_palette()` and `visNetwork::visNetwork(nodes = legend_nodes())`

Examples

```
drake_palette()
visNetwork::visNetwork(nodes = legend_nodes())
```

drake_tip *Function drake_tip*

Description

Output a random tip about drake. Tips are usually related to news and usage.

Usage

```
drake_tip()
```

Examples

```
drake_tip()
cat(drake_tip())
```

evaluate *Function* evaluate

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.

Usage

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

Arguments

plan	workflow plan data frame, similar to one produced by ink{workplan}
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

```
datasets <- workplan(
  small = simulate(5),
  large = simulate(50))
methods <- workplan(
  regression1 = reg1(..dataset..),
  regression2 = reg2(..dataset..))
evaluate(methods, wildcard = "..dataset..",
  values = datasets$target)
evaluate(methods, wildcard = "..dataset..",
  values = datasets$target, expand = FALSE)
x <- workplan(draws = rnorm(mean = Mean, sd = Sd))
evaluate(x, rules = list(Mean = 1:3, Sd = c(1, 10)))
```

---

examples_drake *Function* examples_drake

Description

List the names of all the drake examples. The `'basic'` example is the one from the quickstart vignette: `vignette('quickstart')`.

Usage

```
examples_drake()
```

Value

names of all the drake examples.

See Also

`example_drake`, `make`

Examples

```
examples_drake()
```

example_drake *Function* example_drake

Description

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

Usage

```
example_drake(example = drake::examples_drake(), destination = getwd())
```

Arguments

`example`	name of the example. To see all the available example names, run `examples_drake`.
`destination`	character scalar, file path, where to write the folder containing the code files for the example.

See Also

`examples_drake`, `make`

Examples

```
## Not run:
example_drake('basic') # Same as the quickstart vignette

## End(Not run)
```

expand *Function* expand

Description

Expands a workflow plan data frame by duplicating rows. This generates multiple replicates of targets with the same commands.

Usage

```
expand(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

Examples

```
datasets <- workplan(
  small = simulate(5),
  large = simulate(50))
expand(datasets, values = c("rep1", "rep2", "rep3"))
```

---

failed *Function* failed

Description

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

Examples

```
## Not run:
load_basic_example()
make(my_plan)
failed() # nothing
bad_plan <- workplan(x = function_doesnt_exist())
make(bad_plan) # error
failed() # "x"
diagnose(x)

## End(Not run)
```

`find_cache` *Function* `find_cache`

Description

Return the file path of the nearest drake cache (searching upwards for directories containing a drake cache). Only works if the cache is a file system in a hidden folder named `.drake`.

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

`workplan`, `make`,

Examples

```
## Not run:
load_basic_example()
make(my_plan)
find_cache()

## End(Not run)
```

find_project *Function* find_project

Description

Return the file path of the nearest drake project (searching upwards for directories containing a drake cache). Only works if the cache is a file system in a folder named `.drake`.

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

workplan, make

Examples

```
## Not run:
load_basic_example()
make(my_plan)
find_project()

## End(Not run)
```

gather *Function* gather

Description

Create 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 = 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

workflow plan data frame for aggregating prespecified targets

Examples

```
datasets <- workplan(
  small = simulate(5),
  large = simulate(50))
gather(datasets, target = "my_datasets")
gather(datasets, target = "aggregated_data", gather = "rbind")
```

---

get_cache *Function get_cache*

Description

Search for and return a drake file system cache.

Usage

```
get_cache(path = getwd(), search = TRUE, verbose = TRUE)
```

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

See Also

this_cache, new_cache, recover_cache, config

Examples

```
## Not run:
get_cache()
load_basic_example()
make(my_plan)
x <- get_cache()
x$list()

## End(Not run)
```

`imported` *Function* `imported`

Description

List all the imported objects in the drake cache

Usage

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

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

Value

character vector naming the imported objects in the cache

See Also

`cached`, `loadd`, `built`

Examples

```
## Not run:
load_basic_example()
make(my_plan)
imported()

## End(Not run)
```

`in_memory_cache_types` *Function in_memory_cache_types*

Description

List the supported drake in-memory cache types.

Usage

```
in_memory_cache_types()
```

See Also

`cache_types`, `new_cache`, `get_cache`, `default_cache_type`

Examples

```
cache_types()
```

`in_progress` *Function* `in_progress`

Description

List the targets that either (1) are currently being built if `make()` is running, or (2) were in the process of being built if the previous call to `make()` quit unexpectedly.

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, workplan, make

Examples

```
## Not run:
load_basic_example()
make(my_plan) # Kill before targets finish.
in_progress()

## End(Not run)
```

---

knitr_deps *Function knitr_deps*

Description

Find the dependencies of a dynamic report. To enable drake to watch for these dependencies, your workplan plan command to compile this report must make direct use of knitr::knit(). That is, it must look something like knit('your_report.Rmd') in your workflow plan data frame.

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().

See Also

knitr_deps, deps, make, load_basic_example

Examples

```
## Not run:
load_basic_example()
knitr_deps("'report.Rmd'") # Files must be single-quoted
knitr_deps("report.Rmd")
make(my_plan)
knitr_deps("'report.md'") # Work on the Rmd source, not the output.

## End(Not run)
```

---

legend_nodes *Function legend_nodes*

Description

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

Usage

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

Arguments

font_size	font size of the node label text

See Also

drake_palette(), plot_graph(), dataframes_graph()

Examples

```
## Not run:
visNetwork::visNetwork(nodes = legend_nodes())

## End(Not run)
```

loadd *Function* loadd

Description

Load object(s) from the drake cache 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), envir = parent.frame(), jobs = 1, verbose = TRUE)
```

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.
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	whether to print console messages

See Also

cached, built, imported, workplan, make,

Examples

```
## Not run:
load_basic_example()
make(my_plan)
loadd(reg1) # now check ls()
reg1
loadd(small)
small
loadd(list = c("small", "large"), jobs = 2)
loadd(imported_only = TRUE) # load all imported objects and functions
loadd() # load everything, including built targets

## End(Not run)
```

`load_basic_example` *Function* `load_basic_example`

Description

Loads the basic example into your workspace (or the environment you specify). Also writes/overwrites the file `report.Rmd`. For a thorough walkthrough of how to set up this example, see the quickstart vignette: `vignette('quickstart')`. Alternatively, call `example_drake('basic')` to generate an R script that builds up this example step by step.

Usage

```
load_basic_example(envir = parent.frame(), report_file = "report.Rmd",
  overwrite = FALSE, to = report_file)
```

Arguments

`envir`	The environment to load the example into. Defaults to your workspace. For an insulated workspace, set `envir = new.env(parent = globalenv())`.
`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`

Examples

```
## Not run:
load_basic_example()
deps(reg1)
deps(my_plan$command[1])
deps(my_plan$command[4])
plot_graph(my_plan)
make(my_plan)
clean(destroy = TRUE)
unlink('report.Rmd')
```

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

---

long_hash *Function long_hash*

Description

Get the long hash algorithm of a drake cache.

Usage

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

Arguments

cache	drake cache
verbose	whether to print console messages

Details

See ?default_long_hash_algo()

See Also

default_short_hash_algo, default_long_hash_algo

Examples

```
## Not run:
load_basic_example()
config <- make(my_plan)
cache <- config$cache
long_hash(cache)

## End(Not run)
```

make *Function* make

Description

Run your project (build the targets).

Usage

```
make(plan = workplan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, hook = function(code) {
  force(code) }, cache = drake::get_cache(verbose = verbose),
  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(), clear_progress = NULL,
  imports_only = FALSE, timeout = Inf, cpu = timeout, elapsed = timeout,
  retries = 0, return_config = 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 workplan() 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.
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, whether to print progress to the console. Skipped objects are not printed.
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()). 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){cat("Avoiding the code"` is a bad hook.

cache — drake cache as created by `new_cache()`. See also `get_cache()`, `this_cache()`, and `recover_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 `example_drake("basic")`

jobs — number of parallel processes or jobs to run. For `"future_lapply"` parallelism, `jobs` only applies to the imports. See `future::future.options` for environment variables that control the number of `future_lapply()` jobs for building targets. For example, you might use `options(mc.cores = max_jobs)`. See `max_useful_jobs()` or `plot_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.

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.
clear_progress	logical, whether to clear the saved record of progress seen by progress() and in_progress() before anything is imported or built.
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().
cpu	Seconds of cpu time to allow before imposing a timeout on a target. Passed to R.utils::withTimeout().
elapsed	Seconds of elapsed time to allow before imposing a timeout on a target. Passed to R.utils::withTimeout().
retries	Number of retries to execute if the target fails.
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.

See Also

workplan, workplan, backend, plot_graph, max_useful_jobs, shell_file, silencer_hook

Examples

```
## Not run:
load_basic_example()
outdated(my_plan) # Which targets need to be (re)built?
my_jobs = max_useful_jobs(my_plan) # Depends on what is up to date.
make(my_plan, jobs = my_jobs) # Build what needs to be built.
outdated(my_plan) # Everything is up to date.
reg2 = function(d){ # Change one of your functions.
  d$x3 = d$x^3
  lm(y ~ x3, data = d)
}
outdated(my_plan) # Some targets depend on reg2().
plot_graph(my_plan) # See how they fit in an interactive graph.
make(my_plan) # Rebuild just the outdated targets.
outdated(my_plan) # Everything is up to date again.
plot_graph(my_plan) # The colors changed in the graph.
clean() # Start from scratch
make(my_plan, parallelism = "Makefile", jobs = 4)
clean()
```

```
make(my_plan, parallelism = "Makefile", jobs = 4,
  recipe_command = "R -q -e")

## End(Not run)
```

`Makefile_recipe` *Function Makefile_recipe*

Description

See what your `Makefile` recipes will look like in advance.

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`

See Also

`default_recipe_command`, `r_recipe_wildcard`, `make`

Examples

```
Makefile_recipe()
Makefile_recipe(
  target = "this_target",
  recipe_command = "R -e 'R_RECIPE' -q",
  cache_path = "custom_cache"
)
default_recipe_command()
r_recipe_wildcard()
## Not run:
load_basic_example()
# Look at the Makefile generated by the following.
make(my_plan, paralleliem = "Makefile")
# 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
  recipe_command = "R -q -e")
# Same thing:
clean()
make(my_plan, parallelism = "Makefile", jobs = 2,
  recipe_command = "R -q -e 'R_RECIPE'")
clean()
make(my_plan, parallelism = "Makefile", jobs = 2,
  recipe_command = "R -e 'R_RECIPE' -q")

## End(Not run)
```

`make_imports` *Function make_imports*

Description

just make the imports

Usage

```
make_imports(config)
```

Arguments

`config`	a configuration list returned by `config()`

See Also

`make`, `config`

Examples

```
## Not run:
load_basic_example()
con <- config(my_plan)
make_imports(config = con)

## End(Not run)
```

`max_useful_jobs` *Function* `max_useful_jobs`

Description

Get the maximum number of useful jobs in the next call to `make(..., jobs = YOUR_CHOICE)`.

Usage

```
max_useful_jobs(plan = workplan(), from_scratch = FALSE,
  targets = drake::possible_targets(plan), envir = parent.frame(),
  verbose = TRUE, hook = function(code) {     force(code) },
  cache = drake::get_cache(verbose = verbose), jobs = 1,
  parallelism = drake::default_parallelism(), packages = rev(.packages()),
  prework = character(0), config = NULL, imports = c("files", "all",
  "none"))
```

Arguments

`plan`	workflow plan data frame, same as for function `make()`.
`from_scratch`	logical, whether to compute the max useful jobs as if the workplan were to run from scratch (with all targets out of date).
`targets`	names of targets to build, same as for function `make()`.
`envir`	environment to import from, same as for function `make()`. `config$envir` is ignored in favor of `envir`.
`verbose`	logical, whether to output messages to the console.
`hook`	same as for `make`
`cache`	optional drake cache. See codenew_cache(). If The `cache` argument is ignored if a non-null `config` argument is supplied.
`jobs`	The `outdated()` function is called internally, and it needs to import objects and examine your input files to see what has been updated. This could take some time, and parallel computing may be needed to speed up the process. The `jobs` argument is number of parallel jobs to use for faster computation.
`parallelism`	Choice of parallel backend to speed up the computation. Execution order in `make()` is slightly different when `parallelism` equals `'Makefile'` because in that case, all the imports are imported before any target is built. Thus, `max_useful_jobs()` may give a different answer for Makefile parallelism. See `?parallelism_choices` for details.

packages	same as for make
prework	same as for make
config	internal configuration list of make(...), produced also with config(). config$envir is ignored. Otherwise, if not NULL, config overrides all the other arguments except imports and from_scratch. For example, plan is replaced with config$plan. Computing config in advance could save time if you plan multiple calls to dataframes_graph().
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.

Details

Any additional jobs more than max_useful_jobs(...) will be superfluous, and could even slow you down for make(..., parallelism = 'parLapply'). Set 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 list of three data frames: one for nodes, one for edges, and one for the legend/key nodes.

See Also

plot_graph, build_graph, shell_file

Examples

```
## Not run:
load_basic_example()
plot_graph(my_plan) # Look at the graph to make sense of the output.
max_useful_jobs(my_plan) # 8
max_useful_jobs(my_plan, imports = 'files') # 8
max_useful_jobs(my_plan, imports = 'all') # 10
max_useful_jobs(my_plan, imports = 'none') # 8
make(my_plan)
plot_graph(my_plan)
# Ignore the targets already built.
max_useful_jobs(my_plan) # 1
max_useful_jobs(my_plan, imports = 'files') # 1
max_useful_jobs(my_plan, imports = 'all') # 10
max_useful_jobs(my_plan, 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)
}
plot_graph(my_plan)
max_useful_jobs(my_plan) # 4
max_useful_jobs(my_plan, imports = 'files') # 4
max_useful_jobs(my_plan, imports = 'all') # 10
max_useful_jobs(my_plan, imports = 'none') # 4

## End(Not run)
```

`message_sink_hook` *Function message_sink_hook*

Description

an example hook argument to `make()` that redirects error messages to files.

Usage

```
message_sink_hook(code)
```

Arguments

code	code to run to build the target.

See Also

make, silencer_hook, output_sink_hook

Examples

```
## Not run:
message_sink_hook({
  cat(1234)
  stop(5678)
})
x <- workplan(loud = cat(1234), bad = stop(5678))
make(x, hook = message_sink_hook)

## End(Not run)
```

missed *Function* missed

Description

Report any import objects required by your workplan plan but missing from your workspace. IMPORTANT: you must be in the root directory of your project.

Usage

```
missed(plan = workplan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, jobs = 1,
  parallelism = drake::default_parallelism(), packages = rev(.packages()),
  prework = character(0), config = NULL)
```

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	The outdated() function is called internally, and it needs to import objects and examine your input files to see what has been updated. This could take some time, and parallel computing may be needed to speed up the process. The jobs argument is number of parallel jobs to use for faster computation.
parallelism	Choice of parallel backend to speed up the computation. See ?parallelism_choices for details. The Makefile option is not available here. Drake will try to pick the best option for your system by default.
packages	same as for make
prework	same as for make
config	option internal runtime parameter list of make(...), produced with config(). Overrides all other arguments except if not NULL. For example, config$plan overrides plan. Computing this in advance could save time if you plan multiple calls to missed().

See Also

outdated

Examples

```
## Not run:
load_basic_example()
missed(my_plan)
rm(reg1)
```

```
missed(my_plan)

## End(Not run)
```

---

mk *Function* mk

Description

Internal drake function to be called inside Makefiles only. Makes a single target. Users, do not invoke directly.

Usage

```
mk(target = character(0), cache_path = drake::default_cache_path())
```

Arguments

target	name of target to make
cache_path	path to the drake cache

---

new_cache *Function new_cache*

Description

Make a new drake cache. Could be any type of cache in cache_types().

Usage

```
new_cache(path = drake::default_cache_path(),
  type = drake::default_cache_type(),
  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.
type	character scalar, type of the drake cache. Must be among the list of supported caches in cache_types().
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

See Also

default_short_hash_algo, default_long_hash_algo, make, cache_types, in_memory_cache_types

Examples

```
## Not run:
cache1 <- new_cache() # Creates a new hidden '.drake' folder.
cache2 <- new_cache(path = "not_hidden", short_hash_algo = "md5")

## End(Not run)
e <- new.env()
ls(e)
y <- new_cache(type = "storr_environment", envir = e)
ls(e)
ls(e)
```

---

outdated *Function outdated*

Description

Check which targets are out of date and need to be rebuilt. IMPORTANT: you must be in the root directory of your project.

Usage

```
outdated(plan = workplan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, hook = function(code) {
  force(code) }, cache = drake::get_cache(verbose = verbose),
  parallelism = drake::default_parallelism(), jobs = 1,
  packages = rev(.packages()), prework = character(0), config = NULL)
```

Arguments

plan	same as for make
targets	same as for make
envir	same as for make. Overrides config$envir.
verbose	same as for make
hook	same as for make
cache	optional drake cache. See codenew_cache(). The cache argument is ignored if a non-null config argument is supplied.
parallelism	same as for make
jobs	same as for make
packages	same as for make
prework	same as for make

config — option internal runtime parameter list of make(...), produced with config(). config$envir is ignored. Overrides all the other arguments if not NULL. For example, plan is replaced with config$plan. Computing config in advance could save time if you plan multiple calls to outdated().

See Also

missed, workplan, make, plot_graph

Examples

```
## Not run:
load_basic_example()
outdated(my_plan)
make(my_plan)
outdated(my_plan)

## End(Not run)
```

---

output_sink_hook *Function output_sink_hook*

Description

an example hook argument to make() that redirects output messages to files.

Usage

```
output_sink_hook(code)
```

Arguments

code — code to run to build the target.

See Also

make, silencer_hook, message_sink_hook

Examples

```
## Not run:
output_sink_hook({
  cat(1234)
  stop(5678)
})
x <- workplan(loud = cat(1234), bad = stop(5678))
make(x, hook = output_sink_hook)

## End(Not run)
```

`parallelism_choices` *Function* `parallelism_choices`

Description

List the types of supported parallel computing.

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 `backend()` (or equivalently, `future::plan()`). Then, apply the backend to your workplan 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. for the max number of jobs to use for building targets, use 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

```
parallelism_choices()
parallelism_choices(distributed_only = TRUE)
```

---

plan *Deprecated function* plan

Description

Turns a named collection of command/target pairs into a workflow plan data frame for make and check. Deprecated in favor of workplan() due to a name conflict with future::plan().

Usage

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

Arguments

...	commands named by the targets they generate. Recall that drake uses single quotes to denote external files and double-quoted strings as ordinary strings. Use the strings_in_dots argument to control the quoting in ....
list	character vector of commands named by the targets they generate.
file_targets	logical. If TRUE, targets are single-quoted to tell drake that these are external files that should be expected as output in the next call to make().
strings_in_dots	character scalar. If "filenames", all character strings in ... will be treated as names of file dependencies (single-quoted). If "literals", all character strings in ... will be treated as ordinary strings, not dependencies of any sort (double-quoted). Because of R's automatic parsing/deparsing behavior, strings in ... cannot simply be left alone.

Value

data frame of targets and command

See Also

workplan, make, check

Examples

```
# plan() is deprecated. Use workplan() instead.
workplan(small = simulate(5), large = simulate(50))
workplan(list = c(x = "1 + 1", y = "sqrt(x)"))
workplan(data = readRDS("my_data.rds"))
workplan(
  my_file.rds = saveRDS(1+1, "my_file.rds"),
  file_targets = TRUE,
  strings_in_dots = "literals"
)
```

---

plot_graph *Function* plot_graph

Description

Plot the dependency structure of your workplan. IMPORTANT: you must be in the root directory of your project. To save time for repeated plotting, this function is divided into dataframes_graph() and render_graph().

Usage

```
plot_graph(plan = workplan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, hook = function(code) {
  force(code) }, cache = drake::get_cache(verbose = verbose), jobs = 1,
  parallelism = drake::default_parallelism(), packages = rev(.packages()),
  prework = character(0), config = NULL, 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,
  from_scratch = FALSE, ...)
```

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(). config$envir is ignored in favor of envir.
verbose	logical, whether to output messages to the console.
hook	same as for make
cache	optional drake cache. See codenew_cache(). If The cache argument is ignored if a non-null config argument is supplied.

jobs	The outdated() function is called internally, and it needs to import objects and examine your input files to see what has been updated. This could take some time, and parallel computing may be needed to speed up the process. The jobs argument is number of parallel jobs to use for faster computation.
parallelism	Choice of parallel backend to speed up the computation. Execution order in make() is slightly different when parallelism equals 'Makefile' because in that case, all the imports are imported before any target is built. Thus, the arrangement in the graph is different for Makefile parallelism. See ?parallelism_choices for details.
packages	same as for make().
prework	same as for make().
config	option internal runtime parameter list of make(...), produced with config(). Config overrides all arguments except file, selfcontained, build_times, digits, targets_only, split_columns, font_size, layout, main, direction, hover, navigationButtons, and .... Computing config in advance could save time if you plan multiple calls to outdated().
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
from_scratch	logical, whether to assume that all targets are out of date and the next make() will happen from scratch. Setting to TRUE will prevent the graph from showing you which targets are up to date, but it makes computing the graph much faster.
...	other arguments passed to visNetwork::visNetwork() to plot the graph.

See Also

build_graph

Examples

```
## Not run:
load_basic_example()
plot_graph(my_plan, width = '100%') # The width is passed to visNetwork
make(my_plan)
plot_graph(my_plan) # The red nodes from before are now green.
plot_graph( # plot a subgraph
  my_plan,
  from = c("small", "reg2"),
  to = "summ_regression2_small"
)

## End(Not run)
```

---

possible_targets *Function* possible_targets

Description

internal function, returns the list of possible targets that you can select with the targets argument to make().

Usage

```
possible_targets(plan = workplan())
```

Arguments

plan	workflow plan data frame

Value

character vector of possible targets

See Also

make

Examples

```
## Not run:
load_basic_example()
possible_targets(my_plan)

## End(Not run)
```

---

predict_runtime *Function predict_runtime*

Description

Predict the elapsed runtime of the next call to 'make()'. This function simply sums the elapsed build times. from rate_limiting_times().

Usage

```
predict_runtime(plan = workplan(), from_scratch = FALSE,
  targets_only = FALSE, targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, hook = function(code) {
  force(code) }, cache = drake::get_cache(path = path, search = search,
  verbose = verbose), parallelism = drake::default_parallelism(), jobs = 1,
  future_jobs = jobs, packages = rev(.packages()), prework = character(0),
  config = NULL, digits = 3, path = getwd(), search = TRUE)
```

Arguments

plan	same as for make
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

targets	Targets to build in the workplan. Timing information is limited to these targets and their dependencies.
envir	same as for make
verbose	same as for make
hook	same as for make
cache	optional drake cache. See codenew_cache(). The cache argument is ignored if a non-null config argument is supplied.
parallelism	same as for make. Used to parallelize import objects.
jobs	same as for make, just used to process imports
future_jobs	hypothetical number of jobs assumed for the predicted runtime. assuming this number of jobs.
packages	same as for make
prework	same as for make
config	option internal runtime parameter list of make(...), produced with config(). Overrides all arguments except from_scratch, targets_only, and digits. Computing config in advance could save time if you plan multiple calls to this function.
digits	number of digits for rounding the times.
path	file path to the folder containing the cache. Yes, this is the parent directory containing the cache, not the cache itself.
search	logical, whether to search back in the file system for the cache.

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.

See Also

rate_limiting_times, build_times make

Examples

```
## Not run:
load_basic_example()
make(my_plan)
predict_runtime(my_plan, digits = 4) # everything is up to date
predict_runtime(my_plan, digits = 4, from_scratch = TRUE) # 1 job
predict_runtime(my_plan, future_jobs = 2, digits = 4, from_scratch = TRUE)
predict_runtime(
  my_plan,
  targets = c("small", "large"),
  future_jobs = 2,
  digits = 4,
  from_scratch = TRUE
```

```
)

## End(Not run)
```

---

progress *Function* progress

Description

Get the build progress (overall or individual targets) of the last call to make(). 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)
```

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

Value

Statuses of targets

See Also

diagnose, session, built, imported, readd, workplan, make

Examples

```
## Not run:
load_basic_example()
make(my_plan)
progress()
progress(small, large)
progress(list = c("small", "large"))
progress(no_imported_objects = TRUE)

## End(Not run)
```

---

prune *Deprecated function* prune

Description

Use clean() instead

Usage

```
prune(plan = workplan())
```

Arguments

plan	workflow plan data frame, as generated by workplan.

See Also

clean, make

Examples

```
## Not run:
load_basic_example()
make(my_plan)
cached()
prune(my_plan[1:3,])
cached()
make(my_plan)
clean(destroy = TRUE)

## End(Not run)
```

rate_limiting_times *Function rate_limiting_times*

Description

Return a data frame of elapsed build times of the rate-limiting targets of a `make()` workplan.

Usage

```
rate_limiting_times(plan = workplan(), from_scratch = FALSE,
  targets_only = FALSE, targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, hook = function(code) {
  force(code) }, cache = drake::get_cache(path = path, search = search,
  verbose = verbose), parallelism = drake::default_parallelism(), jobs = 1,
  future_jobs = jobs, packages = rev(.packages()), prework = character(0),
  config = NULL, digits = 3, path = getwd(), search = TRUE)
```

Arguments

plan	same as for make
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.
targets	Targets to build in the workplan. Timing information is limited to these targets and their dependencies.
envir	same as for make. Supersedes config$envir.
verbose	same as for make
hook	same as for make
cache	optional drake cache. See codenew_cache(). The cache argument is ignored if a non-null config argument is supplied.
parallelism	same as for make, and also used to process imported objects/files.
jobs	same as for make, just used to process imports.
future_jobs	hypothetical number of jobs assumed for the predicted runtime. assuming this number of jobs.
packages	same as for make
prework	same as for make
config	option internal runtime parameter list of make(...), produced with config(). Overrides all arguments except from_scratch, targets_only, and digits. Computing config in advance could save time if you plan multiple calls to this function.
digits	number of digits for rounding the times.
path	file path to the folder containing the cache. Yes, this is the parent directory containing the cache, not the cache itself.
search	logical, whether to search back in the file system for the cache.

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.

See Also

`predict_runtime`, `build_times` `make`

Examples

```
## Not run:
load_basic_example()
make(my_plan)
rate_limiting_times(my_plan) # everything is up to date
rate_limiting_times(my_plan, from_scratch  = TRUE, digits = 4) # 1 job
rate_limiting_times(
  my_plan,
  future_jobs = 2,
  from_scratch = TRUE,
  digits = 4
)
rate_limiting_times(
  my_plan,
  targets = c("small", "large"),
  future_jobs = 2,
  from_scratch = TRUE,
  digits = 4
)

## End(Not run)
```

`readd` *Function* `readd`

Description

Read a drake target object from the cache. 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),
  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.
verbose	whether to print console messages

Value

drake target item from the cache

See Also

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

Examples

```
## Not run:
load_basic_example()
make(my_plan)
readd(reg1)
readd(small)
readd("large", character_only = TRUE)
readd("'report.md'") # just a fingerprint of the file (md5 sum)

## End(Not run)
```

read_config *Function* read_config

Description

Read all the configuration parameters from your last attempted call to make(). These include the workflow plan

Usage

```
read_config(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	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 named list of configuration items

See Also

make

Examples

```
## Not run:
load_basic_example()
make(my_plan)
read_config()

## End(Not run)
```

read_graph *Function* read_graph

Description

Read the igraph-style dependency graph of your targets from your last attempted call to make(). For better graphing utilities, see plot_graph() and related functions.

Usage

```
read_graph(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	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 plot_graph()

Value

either a plot or an igraph object, depending on plot

See Also

plot_graph, read_config

Examples

```
## Not run:
load_basic_example()
make(my_plan)
g <- read_graph(plot = FALSE)
class(g)
read_graph() # Actually plot the graph as an interactive visNetwork widget.

## End(Not run)
```

read_plan *Function* read_plan

Description

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

Usage

```
read_plan(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	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_config

Examples

```
## Not run:
load_basic_example()
make(my_plan)
read_plan()

## End(Not run)
```

recover_cache *Function recover_cache*

Description

Load an existing drake files system cache if it exists and create a new one otherwise. Do not use for in-memory caches such as `storr_environment()`.

Usage

```
recover_cache(path = drake::default_cache_path(),
  type = drake::default_cache_type(),
  short_hash_algo = drake::default_short_hash_algo(),
  long_hash_algo = drake::default_long_hash_algo())
```

Arguments

path	file path of the cache
type	character scalar, type of the drake cache. Must be among the list of supported caches in `cache_types()`.
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()`

See Also

`new_cache`, `this_cache`, `get_cache`

Examples

```
## Not run:
load_basic_example()
make(my_plan)
x <- recover_cache(".drake")

## End(Not run)
```

render_graph *Function* render_graph

Description

render a graph from the data frames generated by dataframes_graph()

Usage

```
render_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().

Examples

```
## Not run:
load_basic_example()
graph = dataframes_graph(my_plan)
render_graph(graph, width = '100%') # The width is passed to visNetwork

## End(Not run)
```

`r_recipe_wildcard` *r_recipe_wildcard*

Description

Function to give the R recipe wildcard for Makefiles.

Usage

```
r_recipe_wildcard()
```

Details

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

See Also

`default_recipe_command`

`session` *Function* `session`

Description

Load the `sessionInfo()` of the last call to `make()`.

Usage

```
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`, `workplan`, `make`

Examples

```
## Not run:
load_basic_example()
make(my_plan)
session()

## End(Not run)
```

`shell_file` *Function shell_file*

Description

Write an example `shell.sh` file required by `make(..., parallelism = 'Makefile', prepend = 'SHELL=./shell.sh')` and do a 'chmod +x' to enable execution. Use this option to run your project in parallel on a computing cluster or supercomputer.

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

See Also

`make`, `max_useful_jobs`, `parallelism_choices`

short_hash *Function short_hash*

Description

Get the short hash algorithm of a drake cache.

Usage

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

Arguments

cache	drake cache
verbose	whether to print console messages

Details

See ?default_long_hash_algo()

See Also

default_short_hash_algo, default_long_hash_algo

Examples

```
## Not run:
load_basic_example()
config <- make(my_plan)
cache <- config$cache
short_hash(cache)

## End(Not run)
```

silencer_hook *Function silencer_hook*

Description

an example hook argument to make() that redirects output and error messages

Usage

```
silencer_hook(code)
```

Arguments

code	code to run to build the target.

See Also

make, message_sink_hook, output_sink_hook

Examples

```
## Not run:
silencer_hook({
  cat(1234)
  stop(5678)
})
x <- workplan(loud = cat(1234), bad = stop(5678))
make(x, hook = silencer_hook)

## End(Not run)
```

---

status *Deprecated function* status

Description

Use progress() instead. Gets the build progress (overall or individual targets) of the last call to make(). Objects that drake imported, built, or attempted to build are listed as "finished" or "in progress". Skipped objects are not listed.

Usage

```
status(..., list = character(0), no_imported_objects = FALSE,
  imported_files_only = logical(0), path = getwd(), search = TRUE)
```

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.

Value

Either the build progress of each target given (from the last call to make() or make()), or if no targets are specified, a data frame containing the build progress of the last session. In the latter case, only finished targets are listed.

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

progress, built, imported, readd, workplan, make

Examples

```
## Not run:
load_basic_example()
make(my_plan)
status() # Deprecated. Use progress() instead.
status(small, large)
status(list = c("small", "large"))
status(no_imported_objects = TRUE)

## End(Not run)
```

---

summaries *Function* summaries

Description

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

Usage

```
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

analyses, make, workplan

Examples

```
datasets <- workplan(
  small = simulate(5),
  large = simulate(50))
methods <- workplan(
  regression1 = reg1(..dataset..),
  regression2 = reg2(..dataset..))
analyses <- analyses(methods, datasets = datasets)
summary_types <- workplan(
  summ = summary(..analysis..),
  coef = coefficients(..analysis..))
summaries(summary_types, analyses, datasets, gather = NULL)
summaries(summary_types, analyses, datasets)
summaries(summary_types, analyses, datasets, gather = "list")
summaries(summary_types, analyses, datasets, gather = c("list", "rbind"))
```

---

this_cache *Function this_cache*

Description

Get a known drake file system cache at the exact specified file path Do not use for in-memory caches such as `storr_environment()`.

Usage

```
this_cache(path = drake::default_cache_path())
```

Arguments

path	file path of the cache

Examples

```
## Not run:
x <- this_cache() # Does not exist yet
load_basic_example()
make(my_plan)
y <- this_cache()
z <- this_cache(".drake") # same as above
```

```
manual <- new_cache("manual_cache")
manual2 <- get_cache("manual_cache") # same as above

## End(Not run)
```

`tracked` *Function* `tracked`

Description

Print out which objects, functions, files, targets, etc. are reproducibly tracked.

Usage

```
tracked(plan = workplan(), 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.

Examples

```
## Not run:
load_basic_example()
tracked(my_plan)

## End(Not run)
```

`type_of_cache` *experimental function type_of_cache*

Description

Try to get the type of a drake file system cache. Only works on known caches with known file systems.

Usage

```
type_of_cache(path = drake::default_cache_path())
```

Arguments

path	path to the cache

Details

Experimental function for a possible new feature in the future. It will come in handy if/when multiple cache types are supported.

Examples

```
## Not run:
load_basic_example()
make(my_plan)
type_of_cache(".drake")

## End(Not run)
```

---

workflow *Function* workflow

Description

Turns a named collection of command/target pairs into a workflow plan data frame for make and check.

Usage

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

Arguments

...	same as for drake::workplan()
list	same as for drake::workplan()
file_targets	same as for drake::workplan()
strings_in_dots	same as for drake::workplan()

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

data frame of targets and command

Examples

```
# workflow() is deprecated. Use workplan() instead.
workplan(small = simulate(5), large = simulate(50))
workplan(list = c(x = "1 + 1", y = "sqrt(x)"))
workplan(data = readRDS("my_data.rds"))
workplan(my_file.rds = saveRDS(1+1, "my_file.rds"), file_targets = TRUE,
  strings_in_dots = "literals")
```

---

workplan *Function* workplan

Description

Turns a named collection of command/target pairs into a workflow plan data frame for make and check.

Usage

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

Arguments

...	same as for drake::workplan()
list	same as for drake::workplan()
file_targets	same as for drake::workplan()
strings_in_dots	same as for drake::workplan()

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

data frame of targets and command

Examples

```
workplan(small = simulate(5), large = simulate(50))
workplan(list = c(x = "1 + 1", y = "sqrt(x)"))
workplan(data = readRDS("my_data.rds"))
workplan(my_file.rds = saveRDS(1+1, "my_file.rds"), file_targets = TRUE,
  strings_in_dots = "literals")
```

Index