

Package ‘drake’

October 18, 2017

Title Data Frames in R for Make

Version 4.3.0

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

License GPL-3

Depends R (>= 3.2.0)

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

Suggests abind, 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 2017-10-18 03:15:31 UTC

R topics documented:

drake-package	3
analyses	4
as_file	5
available_hash_algos	6

build_graph	6
build_times	7
built	8
cached	9
cache_path	10
cache_types	10
check	11
clean	12
config	13
configure_cache	14
dataframes_graph	15
default_cache_path	17
default_cache_type	17
default_graph_title	18
default_long_hash_algo	18
default_Makefile_command	19
default_parallelism	20
default_recipe_command	20
default_short_hash_algo	21
default_system2_args	22
deps	22
drake_palette	23
drake_tip	24
evaluate	24
examples_drake	25
example_drake	26
expand	26
find_cache	27
find_project	28
gather	28
get_cache	29
imported	30
in_memory_cache_types	31
in_progress	31
knitr_deps	32
legend_nodes	33
loadd	33
load_basic_example	34
long_hash	35
make	36
Makefile_recipe	38
max_useful_jobs	40
missed	42
mk	43
new_cache	43
outdated	44
parallelism_choices	45
plan	46

plot_graph	48
possible_targets	50
predict_runtime	50
progress	52
prune	53
rate_limiting_times	54
readd	55
read_config	56
read_graph	57
read_plan	58
recover_cache	59
render_graph	60
r_recipe_wildcard	61
session	61
shell_file	62
short_hash	63
status	63
summaries	65
this_cache	66
tracked	66
type_of_cache	67
%>%	68
Index	69

drake-package	<i>Drake is a workflow manager and build system, a scalable solution for reproducibility and high-performance computing.</i>
---------------	--

Description

Drake is a workflow 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.
load(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 <code>..dataset..</code> wildcard where the datasets go. For example, one command could be <code>lm(..dataset..)</code> . Then, the commands in the output will include <code>lm(your_dataset_1)</code> , <code>lm(your_dataset_2)</code> , 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](#), [plan](#)

Examples

```
datasets <- plan(
  small = simulate(5),
  large = simulate(50))
methods <- plan(
  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()
```

build_graph *Function build_graph*

Description

Make a graph of the dependency structure of your workflow.

Usage

```
build_graph(plan = drake::plan(), targets = drake::possible_targets(plan),  
          envir = parent.frame(), 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() .
verbose	logical, whether to output messages to the console.

Details

This function returns an igraph object representing how the targets in your workflow 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	<i>Function</i> 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), targets_only = FALSE)
```

Arguments

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

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	<i>Function</i> 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))
```

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.

Value

list of imported objects in the cache

See Also

[cached](#), [loadadd](#), [link{imported}](#)

Examples

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

cached	<i>Function</i> 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))
```

Arguments

<code>...</code>	objects to load from the cache, as names (unquoted) or character strings (quoted). Similar to <code>...</code> in remove(...) .
<code>list</code>	character vector naming objects to be loaded from the cache. Similar to the <code>list</code> argument of remove() .
<code>no_imported_objects</code>	logical, applies only when no targets are specified and a list of cached targets is returned. If <code>no_imported_objects</code> is TRUE, then <code>cached()</code> 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.
<code>path</code>	Root directory of the drake project, or if <code>search</code> is TRUE, either the project root or a subdirectory of the project. Ignored if a cache is supplied.
<code>search</code>	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.
<code>cache</code>	drake cache. See new_cache() . If supplied, <code>path</code> and <code>search</code> are ignored.

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](#), [plan](#), [make](#)

Examples

```

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

## End(Not run)

```

cache_path	<i>Function cache_path</i>
------------	----------------------------

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	<i>Function cache_types</i>
-------------	-----------------------------

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	<i>Function</i> check
-------	-----------------------

Description

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

Usage

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

Arguments

plan	workflow plan data frame, possibly from plan() .
targets	character vector of targets to make
envir	environment containing user-defined functions
cache	optional drake cache. See new_cache()

Value

invisibly return plan

See Also

[link{plan}](#), [make](#)

Examples

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

clean	<i>Function</i> clean
-------	-----------------------

Description

Cleans up all work done by [make\(\)](#).

Usage

```
clean(..., list = character(0), destroy = FALSE, path = getwd(),
       search = TRUE, cache = drake::get_cache(path = path, search = search))
```

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 cache is supplied, the path and search arguments are ignored.

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 = drake::plan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, cache = drake::get_cache(),
  parallelism = drake::default_parallelism(), jobs = 1,
  packages = (.packages()), prework = character(0),
  prepend = character(0), command = drake::default_Makefile_command(),
  args = drake::default_system2_args(jobs = jobs, verbose = verbose),
  recipe_command = drake::default_recipe_command())

```

Arguments

<code>plan</code>	same as for <code>make</code>
<code>targets</code>	same as for <code>make</code>
<code>envir</code>	same as for <code>make</code>
<code>verbose</code>	same as for <code>make</code>
<code>cache</code>	same as for <code>make</code>
<code>parallelism</code>	same as for <code>make</code>
<code>jobs</code>	same as for <code>make</code>
<code>packages</code>	same as for <code>make</code>
<code>prework</code>	same as for <code>make</code>
<code>prepend</code>	same as for <code>make</code>
<code>command</code>	same as for <code>make</code>
<code>args</code>	same as for <code>make</code>
<code>recipe_command</code>	same as for <code>make</code>

See Also

[plan](#), [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	<i>Function configure_cache</i>
-----------------	---------------------------------

Description

configure a cache for drake. This is to prepare the cache to be called from [make\(\)](#).

Usage

```
configure_cache(cache = drake::get_cache(),
  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)
```

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 <code>digest::digest()</code> . 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 <code>digest::digest()</code> . 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.

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	<i>Function</i> 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 = drake::plan(),
  targets = drake::possible_targets(plan), envir = parent.frame(),
  verbose = TRUE, cache = drake::get_cache(), jobs = 1,
  parallelism = drake::default_parallelism(), packages = (.packages()),
  prework = character(0), build_times = TRUE, digits = 3,
  targets_only = FALSE, split_columns = FALSE, font_size = 20,
  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() . <code>config\$envir</code> is ignored in favor of <code>envir</code> .
verbose	logical, whether to output messages to the console.
cache	optional drake cache. Only used if the <code>config</code> argument is <code>NULL</code> (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.

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)
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(),
  split_columns = FALSE)
```

Arguments

`parallelism` Mode of parallelism intended for the workflow. 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_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 *Function* default_system2_args

Description

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

Usage

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

Arguments

jobs	number of jobs
verbose	logical, whether to be verbose

Value

args for `system2(command, args)`

Examples

```
default_system2_args(jobs = 2, verbose = FALSE)
default_system2_args(jobs = 4, verbose = TRUE)
```

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 `load()` and `read()` 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 <- plan(
  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)
```

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	<i>Function drake_tip</i>
-----------	---------------------------

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	<i>Function evaluate</i>
----------	--------------------------

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

<code>plan</code>	workflow plan data frame, similar to one produced by <code>link{plan}</code>
<code>rules</code>	Named list with wildcards as names and vectors of replacements as values. This is a way to evaluate multiple wildcards at once. When not <code>NULL</code> , <code>rules</code> overrides <code>wildcard</code> and <code>values</code> if not <code>NULL</code> .
<code>wildcard</code>	character scalar denoting a wildcard placeholder
<code>values</code>	vector of values to replace the wildcard in the drake instructions. Will be treated as a character vector. Must be the same length as <code>plan\$command</code> if <code>expand</code> is <code>TRUE</code> .
<code>expand</code>	If <code>TRUE</code> , create a new rows in the workflow plan data frame if multiple values are assigned to a single wildcard. If <code>FALSE</code> , each occurrence of the wildcard is replaced with the next entry in the <code>values</code> 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 <- plan(
  small = simulate(5),
  large = simulate(50))
methods <- plan(
  regression1 = reg1(..dataset..),
  regression2 = reg2(..dataset..))
evaluate(methods, wildcard = "..dataset..",
  values = datasets$target)
evaluate(methods, wildcard = "..dataset..",
  values = datasets$target, expand = FALSE)
x = plan(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	<i>Function</i> 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	<i>Function</i> 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 <- plan(
  small = simulate(5),
  large = simulate(50))
expand(datasets, values = c("rep1", "rep2", "rep3"))
```

find_cache	<i>Function</i> 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

[plan](#), [make](#),

Examples

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

## End(Not run)
```

find_project	<i>Function</i> 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

[plan](#), [make](#)

Examples

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

gather	<i>Function</i> 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 <code>list(...)</code> , <code>c(...)</code> , <code>rbind(...)</code> , or similar.

Value

workflow plan data frame for aggregating prespecified targets

Examples

```
datasets <- plan(  
  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)
```

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 <code>‘.drake‘</code> folder. If you are looking for a different cache with a known folder different from <code>‘.drake‘</code> , use the this_cache() function.
search	logical, whether to search back in the file system for 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))
```

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.

Value

character vector naming the imported objects in the cache

See Also

[cached](#), [loadadd](#), [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))
```

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 <code>codenew_cache()</code> . If cache is supplied, the path and search arguments are ignored.

Value

A character vector of target names

See Also

[session](#), [built](#), [imported](#), [readd](#), [plan](#), [make](#)

Examples

```
## Not run:
load_basic_example()
make(my_plan)
in_progress() # nothing
bad_plan = plan(x = function_doesnt_exist())
make(bad_plan) # error
in_progress() # "x"

## 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 workflow 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 [loadadd\(\)](#) 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	<i>Function legend_nodes</i>
--------------	------------------------------

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	<i>Function loadd</i>
-------	-----------------------

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),  
       envir = parent.frame(), jobs = 1)
```

Arguments

...	targets to load from the cache, as names (unquoted) or character strings (quoted). Similar to ... in <code>remove(...)</code> .
list	character vector naming targets to be loaded from the cache. Similar to the list argument of <code>remove()</code> .
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 <code>codenew_cache()</code> . 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 <code>parallel::mclapply()</code> . Just set jobs to be an integer greater than 1. On Windows, jobs is automatically demoted to 1.

See Also

[cached](#), [built](#), [imported](#), [plan](#), [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(), to = getwd(), overwrite = TRUE)
```

Arguments

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

`to` where to write the dynamic report source file `report.Rmd`

`overwrite` logical, whether to overwrite an existing 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())
```

Arguments

`cache` drake cache

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 = drake::plan(), targets = drake::possible_targets(plan),
      envir = parent.frame(), verbose = TRUE, cache = drake::get_cache(),
      parallelism = drake::default_parallelism(), jobs = 1,
      packages = (.packages()), prework = character(0),
      prepend = character(0), command = drake::default_Makefile_command(),
      args = drake::default_system2_args(jobs = jobs, verbose = verbose),
      recipe_command = drake::default_recipe_command(), return_config = NULL,
      clear_progress = TRUE, imports_only = FALSE)
```

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 plan() to generate workflow plan data frames easily, and see functions analyses() , summaries() , evaluate() , expand() , and gather() for easy ways to generate large workflow plan data frames.
targets	character string, names of targets to build. Dependencies are built too.
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 <code>envir</code> is made, so you don't need to worry about your workspace being modified by <code>make</code> . The deep copy inherits from the global environment. Wherever necessary, objects and functions are imported from <code>envir</code> and the global environment and then reproducibly tracked as dependencies.
verbose	logical, whether to print progress to the console. Skipped objects are not printed.
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 <code>parallelism_choices()</code> . For detailed explanations, see <code>?parallelism_choices</code> , the tutorial vignettes, or the tutorial files generated by <code>example_drake("basic")</code>
jobs	number of parallel processes or jobs to run. See <code>max_useful_jobs()</code> or <code>plot_graph()</code> to help figure out what the number of jobs should be. Windows users should not set <code>jobs > 1</code> if <code>parallelism</code> is "mclapply" because <code>mclapply()</code> is based on forking. Windows users who use <code>parallelism == "Makefile"</code> will need to download and install Rtools. If <code>parallelism</code> 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 <code>jobs</code> argument to <code>make()</code> . To use at most 2 jobs for imports and at most 4 jobs for targets, run <code>make(..., parallelism = "Makefile", jobs = 2, args</code>
packages	character vector packages to load, in the order they should be loaded. Defaults to <code>(.packages())</code> , so you should not usually need to set this manually. Just call <code>library()</code> to load your packages before <code>make()</code> . However, sometimes packages need to be strictly forced to load in a certain order, especially if <code>parallelism</code> is "Makefile". To do this, do not use <code>library()</code> or <code>require()</code> or <code>loadNamespace()</code> or <code>attachNamespace()</code> to load any libraries beforehand. Just list your packages in the <code>packages</code> argument in the order you want them to be loaded. If <code>parallelism</code> is "mclapply", the necessary packages are loaded once before any targets are built. If <code>parallelism</code> 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 <code>packages</code> argument are loaded before any prework is done. If <code>parallelism</code> is "mclapply", the prework is run once before any targets are built. If <code>parallelism</code> 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 <code>parallelism</code> is "Makefile". See the vignettes (<code>vignette(package = "drake")</code>) to learn how to use <code>prepend</code> to take advantage of multiple nodes of a supercomputer.
command	character scalar, command to call the Makefile generated for distributed computing. Only applies when <code>parallelism</code> is "Makefile". Defaults to the usual "make" (<code>default_Makefile_command()</code>), but it could also be "lsmake" on supporting systems, for example. <code>command</code> and <code>args</code> are executed via <code>system2(command, args)</code> to run the Makefile. If <code>args</code> has something like "--jobs=2", or if <code>jobs >= 2</code> and <code>args</code> 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, <code>jobs</code> and <code>verbose</code> are overwritten as they apply to the Makefile. <code>command</code> and <code>args</code> are executed via <code>system2(command, args)</code> to run the Makefile. If <code>args</code> has something like "--jobs=2", or if <code>jobs >= 2</code> and <code>args</code> 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.

`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.

`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.

See Also

[plan](#), [plot_graph](#), [max_useful_jobs](#), [shell_file](#)

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

<code>recipe_command</code>	The Makefile recipe command. See default_recipe_command() .
<code>target</code>	character scalar, name of your target
<code>cache_path</code>	path to the drake cache. In practice, this defaults to the hidden <code>.drake/</code> folder, but this can be customized. In the Makefile, the drake cache is coded with the Unix variable <code>'DRAKE_CACHE'</code> and then dereferenced with <code>'\$(DRAKE_CACHE)'</code> . To simplify things for users who may be unfamiliar with Unix variables, the <code>recipe()</code> 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")`, # no-lint from within R, the Makefile builds each target with the Makefile recipe, `R -e 'drake::mk("this_target", "path_to_cache")`. 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", "path_to_cache")`.

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, parallelism = "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)

```

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 = drake::plan(), from_scratch = FALSE,
  targets = drake::possible_targets(plan), envir = parent.frame(),
  verbose = TRUE, cache = drake::get_cache(), jobs = 1,
  parallelism = drake::default_parallelism(), packages = (.packages()),
  prework = character(0), config = NULL, imports = c("files", "all",
  "none"))

```

Arguments

<code>plan</code>	workflow plan data frame, same as for function make() .
<code>from_scratch</code>	logical, whether to compute the max useful jobs as if the workflow were to run from scratch (with all targets out of date).
<code>targets</code>	names of targets to build, same as for function make() .
<code>envir</code>	environment to import from, same as for function make() . <code>config\$envir</code> is ignored in favor of <code>envir</code> .
<code>verbose</code>	logical, whether to output messages to the console.
<code>cache</code>	optional drake cache. See codenew_cache() . If The cache argument is ignored if a non-null config argument is supplied.
<code>jobs</code>	The <code>outdated()</code> 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 <code>jobs</code> argument is number of parallel jobs to use for faster computation.
<code>parallelism</code>	Choice of parallel backend to speed up the computation. Execution order in make() is slightly different when <code>parallelism</code> equals 'Makefile' because in that case, all the imports are imported before any target is built. Thus, <code>max_useful_jobs()</code> may give a different answer for Makefile parallelism. See <code>?parallelism_choices</code> for details.
<code>packages</code>	same as for make
<code>prework</code>	same as for make

config	internal configuration list of <code>make(...)</code> , produced also with <code>config()</code> . <code>config\$envir</code> is ignored. Otherwise, if not NULL, <code>config</code> overrides all the other arguments except <code>imports</code> and <code>from_scratch</code> . For example, <code>plan</code> is replaced with <code>config\$plan</code> . Computing <code>config</code> in advance could save time if you plan multiple calls to <code>dataframes_graph()</code> .
imports	Set the <code>imports</code> argument to change your assumptions about how fast objects/files are imported. Possible values: <ul style="list-style-type: none"> • 'all': Factor all imported files/objects into calculating the max useful number of jobs. Note: this is not appropriate for <code>make(..., parallelism = 'Makefile')</code> 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)

```

missed

Function missed

Description

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

Usage

```

missed(plan = drake::plan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, jobs = 1,
  parallelism = drake::default_parallelism(), packages = (.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 <code>outdated()</code> 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 <code>jobs</code> argument is number of parallel jobs to use for faster computation.
parallelism	Choice of parallel backend to speed up the computation. See <code>?parallelism_choices</code> 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, <code>config\$plan</code> overrides <code>plan</code> . Computing this in advance could save time if you plan multiple calls to <code>missed()</code> .

See Also

[outdated](#)

Examples

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

## End(Not run)
```

mk	<i>Function mk</i>
----	--------------------

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	<i>Function new_cache</i>
-----------	---------------------------

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 = drake::plan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, cache = drake::get_cache(),
  parallelism = drake::default_parallelism(), jobs = 1,
  packages = (.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
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](#), [plan](#), [make](#), [plot_graph](#)

Examples

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

parallelism_choices *Function* parallelism_choices

Description

List the types of supported parallel computing.

Usage

```
parallelism_choices()
```

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.

'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()
```

plan

Function plan

Description

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

Usage

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

Arguments

<code>...</code>	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 <code>strings_in_dots</code> argument to control the quoting in <code>...</code>
<code>list</code>	character vector of commands named by the targets they generate.
<code>file_targets</code>	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 <code>make()</code> .
<code>strings_in_dots</code>	character scalar. If "filenames", all character strings in <code>...</code> will be treated as names of file dependencies (single-quoted). If "literals", all character strings in <code>...</code> will be treated as ordinary strings, not dependencies of any sort (double-quoted). Because of R's automatic parsing/deparsing behavior, strings in <code>...</code> cannot simply be left alone.

Details

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

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

Value

data frame of targets and command

See Also

`link{check}`, `make`,

Examples

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

plot_graph	<i>Function</i> plot_graph
------------	----------------------------

Description

Plot the dependency structure of your workflow. **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 = drake::plan(), targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, cache = drake::get_cache(),
  jobs = 1, parallelism = drake::default_parallelism(),
  packages = (.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, ...)
```

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() . <code>config\$envir</code> is ignored in favor of <code>envir</code> .
verbose	logical, whether to output messages to the console.
cache	optional drake cache. See codenew_cache() . If The cache argument is ignored if a non-null <code>config</code> argument is supplied.
jobs	The <code>outdated()</code> 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 <code>jobs</code> 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 <code>parallelism</code> 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 <code>?parallelism_choices</code> for details.
packages	same as for make() .
prework	same as for make() .
config	option internal runtime parameter list of make(...) , produced with config() . <code>Config</code> overrides all arguments except <code>file</code> , <code>selfcontained</code> , <code>build_times</code> , <code>digits</code> , <code>targets_only</code> , <code>split_columns</code> , <code>font_size</code> , <code>layout</code> , <code>main</code> , <code>direction</code> , <code>hover</code> , <code>navigationButtons</code> , and <code>...</code> . Computing <code>config</code> in advance could save time if you plan multiple calls to <code>outdated()</code> .

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 <code>build_times()</code> 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 <code>visNetwork::visHierarchicalLayout()</code> 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 <code>visNetwork::visInteraction(navigationButtons = ...)</code>
...	other arguments passed to <code>visNetwork::visNetwork()</code> 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.

## 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 = drake::plan())
```

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 = drake::plan(), from_scratch = FALSE,
  targets_only = FALSE, targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, cache = drake::get_cache(path =
  path, search = search), parallelism = drake::default_parallelism(),
  jobs = 1, future_jobs = jobs, packages = (.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 workflow. Timing information is limited to these targets and their dependencies.
envir	same as for make
verbose	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))

```

Arguments

<code>...</code>	objects to load from the cache, as names (unquoted) or character strings (quoted). Similar to <code>...</code> in <code>remove(...)</code> .
<code>list</code>	character vector naming objects to be loaded from the cache. Similar to the <code>list</code> argument of <code>remove()</code> .
<code>no_imported_objects</code>	logical, whether to only return information about imported files and targets with commands (i.e. whether to ignore imported objects that are not files).
<code>imported_files_only</code>	logical, deprecated. Same as <code>no_imported_objects</code> . Use the <code>no_imported_objects</code> argument instead.
<code>path</code>	Root directory of the drake project, or if <code>search</code> is <code>TRUE</code> , either the project root or a subdirectory of the project.
<code>search</code>	If <code>TRUE</code> , search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
<code>cache</code>	optional drake cache. See <code>codenew_cache()</code> . If <code>cache</code> is supplied, the <code>path</code> and <code>search</code> arguments are ignored.

Value

Statuses of targets

See Also

[session](#), [built](#), [imported](#), [readd](#), [plan](#), [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 = drake::plan())
```

Arguments

`plan` workflow plan data frame, as generated by [plan](#).

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()` workflow.

Usage

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

Arguments

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

Arguments

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

Value

drake target item from the cache

See Also

[loadadd](#), [cached](#), [built](#), [link{imported}](#), [plan](#), [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))
```

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.

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	<i>Function</i> 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), ...)
```

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 <code>new_cache()</code> . If cache is supplied, the path and search arguments are ignored.
...	arguments to <code>visNetwork()</code> via <code>plot_graph()</code>

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	<i>Function</i> 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))
```

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 <code>new_cache()</code> . If cache is supplied, the path and search arguments are ignored.

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	<i>Function recover_cache</i>
---------------	-------------------------------

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	<i>Function</i> 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, ...)
```

Arguments

graph_dataframes	list of data frames generated by <code>dataframes_graph()</code> . There should be 3 data frames: nodes, edges, and legend_nodes.
file	Name of HTML file to save the graph. If NULL or <code>character(0)</code> , no file is saved and the graph is rendered and displayed within R.
layout	name of an igraph layout to use, such as <code>'layout_with_sugiyama'</code> or <code>'layout_as_tree'</code> . Be careful with <code>'layout_as_tree'</code> : the graph is a directed acyclic graph, but not necessarily a tree.
direction	an argument to <code>visNetwork::visHierarchicalLayout()</code> indicating the direction of the graph. Options include <code>'LR'</code> , <code>'RL'</code> , <code>'DU'</code> , and <code>'UD'</code> . 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 <code>visNetwork::visInteraction(navigationButtons = ...)</code>
...	arguments passed to <code>visNetwork()</code> .

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))
```

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.

Value

[sessionInfo\(\)](#) of the last call to [make\(\)](#)

See Also

[built](#), [imported](#), [readd](#), [plan](#), [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 = file.path(getwd(), "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	<i>Function short_hash</i>
------------	----------------------------

Description

Get the short hash algorithm of a drake cache.

Usage

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

Arguments

cache	drake cache
-------	-------------

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)
```

status	<i>Deprecated function status</i>
--------	-----------------------------------

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 <code>remove(...)</code> .
list	character vector naming objects to be loaded from the cache. Similar to the list argument of <code>remove()</code> .
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 <code>no_imported_objects</code> . Use the <code>no_imported_objects</code> argument instead.
path	Root directory of the drake project, or if <code>search</code> 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 are cached or a character vector listing all cached items, depending on whether any targets are specified

See Also

[progress](#), [built](#), [imported](#), [readd](#), [plan](#), [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	<i>Function summaries</i>
-----------	---------------------------

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 <code>..analysis..</code> and <code>..dataset..</code> wildcards just like the <code>..dataset..</code> 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](#), [plan](#)

Examples

```
datasets <- plan(
  small = simulate(5),
  large = simulate(50))
methods <- plan(
  regression1 = reg1(..dataset..),
  regression2 = reg2(..dataset..))
analyses <- analyses(methods, datasets = datasets)
summary_types <- plan(
  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	<i>Function this_cache</i>
------------	----------------------------

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	<i>Function tracked</i>
---------	-------------------------

Description

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

Usage

```
tracked(plan = drake::plan(), targets = drake::possible_targets(plan),
  envir = parent.frame())
```

Arguments

plan	workflow plan data frame, same as for function <code>make()</code> .
targets	names of targets to build, same as for function <code>make()</code> .
envir	environment to import from, same as for function <code>make()</code> .

Examples

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

type_of_cache	<i>experimental function type_of_cache</i>
---------------	--

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)
```

`%>%`*Pipe operator*

Description

Pipe operator from the `magrittr` package

Usage

```
lhs %>% rhs
```

Arguments

lhs	left-hand side argument
rhs	right-hand side argument

Examples

```
rnorm(2) %>%  
  mean %>%  
  c("third element")
```

Index

- [%>%, 68](#)
- [analyses, 4, 36, 65](#)
- [as_file, 5](#)
- [attachNamespace, 37](#)
- [available_hash_algos, 6, 14, 19, 21](#)
- [build_graph, 6, 16, 41, 49](#)
- [build_times, 7, 16, 49, 51, 55](#)
- [built, 7, 8, 9, 30, 32, 34, 53, 56, 62, 64](#)
- [c, 29](#)
- [cache_path, 10](#)
- [cache_types, 10, 31, 43, 44, 59](#)
- [cached, 8, 9, 30, 34, 56](#)
- [check, 11, 46](#)
- [clean, 12, 19, 21, 53, 54](#)
- [config, 13, 16, 29, 41, 42, 45, 48, 51, 54](#)
- [configure_cache, 14, 18, 21](#)
- [dataframes_graph, 15, 18, 33, 48, 60](#)
- [default_cache_path, 17](#)
- [default_cache_type, 10, 17, 17, 31](#)
- [default_graph_title, 18](#)
- [default_long_hash_algo, 14, 15, 18, 19, 35, 44, 59, 63](#)
- [default_Makefile_command, 19, 37](#)
- [default_parallelism, 20](#)
- [default_recipe_command, 20, 39, 61](#)
- [default_short_hash_algo, 14, 15, 21, 21, 35, 44, 59, 63](#)
- [default_system2_args, 22](#)
- [deps, 22, 32](#)
- [drake \(drake-package\), 3](#)
- [drake-package, 3](#)
- [drake_palette, 23, 33](#)
- [drake_tip, 24](#)
- [evaluate, 24, 36](#)
- [example_drake, 25, 26, 34, 37](#)
- [examples_drake, 25, 26](#)
- [expand, 26, 36](#)
- [find_cache, 27](#)
- [find_project, 28](#)
- [gather, 28, 36, 65](#)
- [get_cache, 10, 17, 29, 31, 36, 59](#)
- [imported, 9, 30, 32, 34, 53, 62, 64](#)
- [in_memory_cache_types, 10, 31, 44](#)
- [in_progress, 31, 38](#)
- [knitr_deps, 32, 32](#)
- [legend_nodes, 23, 33](#)
- [library, 37, 56](#)
- [list, 29](#)
- [load_basic_example, 32, 34](#)
- [load, 8, 9, 22, 30, 32, 33, 56](#)
- [loadNamespace, 37](#)
- [long_hash, 35](#)
- [make, 5, 6, 9, 11–16, 19–21, 25–28, 31, 32, 34, 36, 37, 39–42, 44–48, 50–59, 61–66](#)
- [Makefile_recipe, 20, 38, 61](#)
- [max_useful_jobs, 37, 38, 40, 62](#)
- [mclapply, 37, 46](#)
- [missed, 42, 45](#)
- [mk, 43](#)
- [new_cache, 8–12, 15, 17, 29–31, 34, 36, 40, 43, 45, 48, 51, 52, 54, 56–59, 61](#)
- [outdated, 13, 42, 44](#)
- [parallelism_choices, 18, 37, 45, 62](#)
- [parLapply, 46](#)
- [plan, 5, 9, 11, 14, 27, 28, 32, 34, 36, 38, 45, 46, 53, 56, 62, 64, 65](#)
- [plot.igraph, 6](#)
- [plot_graph, 6, 13, 14, 16, 18, 23, 33, 37, 38, 41, 45, 48, 57, 58](#)

possible_targets, [50](#)
predict_runtime, [50](#), [55](#)
progress, [38](#), [52](#), [63](#), [64](#)
prune, [12](#), [53](#)

r_recipe_wildcard, [39](#), [61](#)
rate_limiting_times, [50](#), [51](#), [54](#)
rbind, [29](#)
read_config, [56](#), [58](#), [59](#)
read_graph, [57](#)
read_plan, [58](#)
readd, [9](#), [22](#), [32](#), [53](#), [55](#), [62](#), [64](#)
recover_cache, [29](#), [36](#), [59](#)
remove, [9](#), [12](#), [34](#), [52](#), [64](#)
render_graph, [48](#), [60](#)
require, [37](#)

session, [32](#), [53](#), [61](#)
sessionInfo, [61](#), [62](#)
shell_file, [20](#), [38](#), [41](#), [46](#), [62](#)
short_hash, [63](#)
status, [63](#)
summaries, [5](#), [36](#), [65](#)
system.time, [7](#)
system2, [22](#), [37](#)

this_cache, [29](#), [36](#), [59](#), [66](#)
tracked, [66](#)
type_of_cache, [67](#)