Package 'drake'

August 4, 2017

Title Data Frames in R for Make

Version 4.0.0

Date 2017-08-03

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

License GPL-3

Depends R (>= 3.2.0)

Imports base64url, codetools, crayon, eply, digest, igraph, magrittr, parallel, plyr, R.utils, stats, storr, stringi, stringr, testthat, tools, utils, visNetwork

Suggests abind, knitr, MASS, 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], Eli Lilly and Company [cph]

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

Repository CRAN

Date/Publication 2017-08-04 20:46:44 UTC

R topics documented:

drake-package *A scalable solution for reproducibility and high-performance computing*

Description

Drake is a workflow manager and build system.

Author(s)

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

References

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

Examples

```
## Not run:
library(drake)
load_basic_example()
make(my_plan) # Build everything.
make(my_plan) # Nothing is done because everything is already up to date.
reg2 = function(d){ # Change one of your functions.
  d$x3 = d$x^3
  lm(y ~ x3, data = d)
}
make(my_plan) # Only the pieces depending on reg2() get rebuilt.
readd(small) # Read/load from the cache.
loadd(large)
head(large)
clean() # Restart from scratch
make(my_plan, jobs = 2) # Distribute over 2 parallel jobs.
clean()
make(my_plan, jobs = 4, parallelism = "Makefile") # Distribute over 4 parallel R sessions.
make(my_plan, jobs = 4, parallelism = "Makefile") # Everything up to date.
clean(destroy = TRUE) # Totally remove the cache.
unlink(c("Makefile", "report.Rmd"))

## End(Not run)
```

analyses *Function* analyses

Description

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

Usage

```
analyses(plan, datasets)
```

Arguments

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

Value

an evaluated workflow plan data frame of analysis instructions

See Also

summaries, make, 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")
```

`build_graph` *Function* `build_graph`

Description

Make a graph of the dependency structure of your workflow.

Usage

```
build_graph(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 bulid, 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)
```

built *Function* built

Description

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

Usage

```
built(path = getwd(), search = TRUE)
```

Arguments

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

Value

list of imported objects in the cache

See Also

cached, loadd, link{imported}

Examples

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

## End(Not run)
```

cached *Function* cached

Description

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

Usage

```
cached(..., list = character(0), no_imported_objects = FALSE,
  path = getwd(), search = TRUE)
```

Arguments

`...`	objects to load from the cache, as names (unquoted) or character strings (quoted). Similar to `...` in `remove(...)`.
`list`	character vector naming objects to be loaded from the cache. Similar to the `list` argument of `remove()`.
`no_imported_objects`	logical, applies only when no targets are specified and a list of cached targets is returned. If `no_imported_objects` is `TRUE`, then `cached()` shows built targets (with commands) plus imported files, ignoring imported objects. Otherwise, the full collection of all cached objects will be listed. Since all your functions and all their global variables are imported, the full list of imported objects could get really cumbersome.
`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project.
`search`	logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.

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)
```

---

check *Function* check

Description

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

Usage

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

Arguments

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

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 *Function* clean

Description

Cleans up all work done by make().

Usage

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

Arguments

...	targets to remove from the cache, as names (unquoted) or character strings (quoted). Similar to ... in remove(...).
list	character vector naming targets to be removed from the cache. Similar to the list argument of remove().

destroy	logical, whether to totally remove the drake cache. If destroy is FALSE, only the targets from make() are removed. If TRUE, the whole cache is removed, including session metadata, etc.
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.

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, return_config = TR in that the graph includes both the targets and the imports, not just the imports.

Usage

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

Arguments

plan	same as for make
targets	same as for make
envir	same as for make
verbose	same as for make
parallelism	same as for make
jobs	same as for make
packages	same as for make
prework	same as for make

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)
```

---

dataframes_graph *Function* dataframes_graph

Description

Get the information about nodes, edges, and the legend/key so you can plot your own custom visNetwork. IMPORTANT: you must be in the root directory of your project.

Usage

```
dataframes_graph(plan, targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, jobs = 1,
  parallelism = drake::default_parallelism(), packages = (.packages()),
  prework = character(0), targets_only = FALSE, font_size = 20,
  config = NULL)
```

Arguments

plan	workflow plan data frame, same as for function make().
targets	names of targets to bulid, same as for function make().
envir	environment to import from, same as for function make().
verbose	logical, whether to output messages to the console.
jobs	The outdated() function is called internally, and it needs to import objects and examine your input files to see what has been updated. This could take some time, and parallel computing may be needed to speed up the process. The jobs argument is number of parallel jobs to use for faster computation.
parallelism	Choice of parallel backend to speed up the computation. See ?parallelism_choices for details. The Makefile option is not available here. Drake will try to pick the best option for your system by default.
packages	same as for make
prework	same as for make
targets_only	logical, whether to skip the imports and only include the targets in the workflow plan.
font_size	numeric, font size of the node labels in the graph
config	option internal runtime parameter list of make(...), produced with config(). Computing this in advance could save time if you plan multiple calls to dataframes_graph().

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

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_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_system2_args` *Internal function* `default_system2_args`

Description

Internal function to configure arguments to `system2()` to run Makefiles. Not a user-side function.

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.

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])
```

drake_tip *Function drake_tip*

Description

Output a random tip about drake.

Usage

```
drake_tip()
```

Examples

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

evaluate *Function* evaluate

Description

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

Usage

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

Arguments

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

Details

Specify a single wildcard with the `wildcard` and `values` arguments. In each command, the text in `wildcard` will be replaced by each value in `values` in turn. Specify multiple wildcards ith 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)
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 *Function* example_drake

Description

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

Usage

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

Arguments

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

See Also

`examples_drake`, `make`

Examples

```
## Not run:
example_drake("basic") # Walkthrough: vignette("quickstart")

## End(Not run)
```

expand *Function* expand

Description

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

Usage

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

Arguments

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

Value

an expanded workflow plan data frame

Examples

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

`find_cache` *Function* `find_cache`

Description

Return the file path of the nearest drake cache (searching upwards for directories containing a drake cache).

Usage

```
find_cache(path = getwd())
```

Arguments

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

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 *Function* find_project

Description

Return the file path of the nearest drake project (searching upwards for directories containing a drake cache).

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 *Function* gather

Description

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

Usage

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

Arguments

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

Value

workflow plan data frame for aggregating prespecified targets

Examples

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

---

imported *Function* imported

Description

List all the imported objects in the drake cache

Usage

```
imported(files_only = FALSE, path = getwd(), search = TRUE)
```

Arguments

files_only	logical, whether to show imported files only and ignore imported objects. Since all your functions and all their global variables are imported, the full list of imported objects could get really cumbersome.
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project.
search	logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.

Value

character vector naming the imported objects in the cache

See Also

cached, loadd, built

Examples

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

## End(Not run)
```

`in_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)
```

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.

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)
```

loadd *Function* loadd

Description

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

Usage

```
loadd(..., list = character(0), imported_only = FALSE, path = getwd(),
  search = TRUE, envir = parent.frame())
```

Arguments

`...`	targets to load from the cache, as names (unquoted) or character strings (quoted). Similar to `...` in `remove(...)`.
`list`	character vector naming targets to be loaded from the cache. Similar to the `list` argument of `remove()`.
`imported_only`	logical, whether only imported objects should be loaded.
`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project.
`search`	logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only.
`envir`	environment to load objects into. Defaults to the calling environment (current workspace).

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"))
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())
```

Arguments

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

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)
```

make *Function* make

Description

Run your project (build the targets).

Usage

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

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 envir is made, so you don't need to worry about your workspace being modified by make. The deep copy inherits from the global environment. Wherever necessary, objects and functions are imported from envir and the global environment and then reproducibly tracked as dependencies.
verbose	logical, whether to print progress to the console. Skipped objects are not printed.
imports_only	logical, whether to skip building the targets in plan and just import objects and files.
parallelism	character, type of parallelism to use. To list the options, call parallelism_choices(). For detailed explanations, see ?parallelism_choices, the tutorial vignettes, or the tutorial files generated by example_drake("basic")
jobs	number of parallel processes or jobs to run. See max_useful_jobs() or plot_graph() to help figure out what the number of jobs should be. Windows users should not set jobs > 1 if parallelism is "mclapply" because mclapply() is based on forking. Windows users who use parallelism == "Makefile" will need to download and install Rtools. If parallelism is "Makefile", Makefile-level parallelism is only used for targets in your workflow plan data frame, not imports. To process imported objects and files, drake selects the best parallel backend for your system and uses the number of jobs you give to the jobs argument to make(). To use at most 2 jobs for imports and at most 4 jobs for targets, run make(..., parallelism = "Makefile", jobs = 2, args
packages	character vector packages to load, in the order they should be loaded. Defaults to (.packages()), so you shouldn't usually need to set this manually. Just call library() to load your packages before make(). However, sometimes packages need to be strictly forced to load in a certian order, especially if parallelism is "Makefile". To do this, do not use library() or require() or loadNamespace() or attachNamespace() to load any libraries beforehand. Just list your packages in the packages argument in the order you want them to be loaded. If parallelism is "mclapply", the necessary packages are loaded once before any targets are built. If parallelism is "Makefile", the necessary packages are loaded once on initialization and then once again for each target right before that target is built.
prework	character vector of lines of code to run before build time. This code can be used to load packages, set options, etc., although the packages in the packages argument are loaded before any prework is done. If parallelism is "mclapply", the prework is run once before any targets are built. If parallelism is "Makefile",

the prework is run once on initialization and then once again for each target right before that target is built.

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

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

args — command line arguments to call the Makefile for distributed computing. For advanced users only. If set, `jobs` and `verbose` are overwritten as they apply to the Makefile. `command` and `args` are executed via `system2(command, args)` to run the Makefile. If `args` has something like `"--jobs=2"`, or if `jobs >= 2` and `args` is left alone, targets will be distributed over independent parallel R sessions wherever possible.

return_config — logical, whether to return the internal list of runtime configuration parameters used by `make()`

clear_progress — logical, whether to clear the saved record of progress seen by `progress()` and `in_progress()` before anything is imported or built.

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.

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

Arguments

`plan`	workflow plan data frame, same as for function `make()`.
`targets`	names of targets to bulid, same as for function `make()`.
`envir`	environment to import from, same as for function `make()`.
`verbose`	logical, whether to output messages to the console.
`jobs`	The `outdated()` function is called internally, and it needs to import objects and examine your input files to see what has been updated. This could take some time, and parallel computing may be needed to speed up the process. The `jobs` argument is number of parallel jobs to use for faster computation.
`parallelism`	Choice of parallel backend to speed up the computation. See `?parallelism_choices` for details. The Makefile option is not available here. Drake will try to pick the best option for your system by default.
`packages`	same as for `make`
`prework`	same as for `make`
`config`	internal configuration list of `make(...)`, produced also with `config()`. Computing this in advance could save time if you plan multiple calls to `dataframes_graph()`.
`imports`	Set the `imports` argument to change your assumptions about how fast objects/files are imported. Possible values: • "all": Factor all imported files/objects into calculating the max useful number of jobs. Note: this is not appropriate for `make(.., parallelism = "Makefile")` because imports are processed sequentially for the Makefile option. • "files": Factor all imported files into the calculation, but ignore all the other imports. • "none": Ignore all the imports and just focus on the max number of useful jobs for parallelizing targets.

Details

Any additional jobs more than `max_useful_jobs(...)` will be superfluous, and could even slow you down for `make(..., parallelism = "parLapply")`. Set Set the `imports` argument to change your assumptions about how fast objects/files are imported. IMPORTANT: you must be in the root directory of your project.

Value

a list of three data frames: one for nodes, one for edges, and one for the legend/key nodes.

See Also

`plot_graph`, `build_graph`, `shell_file`

Examples

```
## Not run:
load_basic_example()
plot_graph(my_plan) # Look at the graph to make sense of the output.
max_useful_jobs(my_plan) # 8
max_useful_jobs(my_plan, imports = "files") # 8
max_useful_jobs(my_plan, imports = "all") # 10
max_useful_jobs(my_plan, imports = "none") # 8
make(my_plan)
plot_graph(my_plan)
# Ignore the targets already built.
max_useful_jobs(my_plan) # 1
max_useful_jobs(my_plan, imports = "files") # 1
max_useful_jobs(my_plan, imports = "all") # 10
max_useful_jobs(my_plan, imports = "none") # 0
# Change a function so some targets are now out of date.
reg2 = function(d){
  d$x3 = d$x^3
  lm(y ~ x3, data = d)
}
plot_graph(my_plan)
max_useful_jobs(my_plan) # 4
max_useful_jobs(my_plan, imports = "files") # 4
max_useful_jobs(my_plan, imports = "all") # 10
max_useful_jobs(my_plan, imports = "none") # 4

## End(Not run)
```

`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, targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, jobs = 1,
  parallelism = drake::default_parallelism(), packages = (.packages()),
  prework = character(0), font_size = 20, config = NULL)
```

Arguments

plan	workflow plan data frame, same as for function make().
targets	names of targets to bulid, same as for function make().
envir	environment to import from, same as for function make().
verbose	logical, whether to output messages to the console.
jobs	The outdated() function is called internally, and it needs to import objects and examine your input files to see what has been updated. This could take some time, and parallel computing may be needed to speed up the process. The jobs argument is number of parallel jobs to use for faster computation.
parallelism	Choice of parallel backend to speed up the computation. See ?parallelism_choices for details. The Makefile option is not available here. Drake will try to pick the best option for your system by default.
packages	same as for make
prework	same as for make
font_size	numeric, font size of the node labels in the graph
config	option internal runtime parameter list of make(...), produced with config(). Computing this in advance could save time if you plan multiple calls to missed().

See Also

outdated

Examples

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

## End(Not run)
```

mk *Function* mk

Description

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

Usage

```
mk(target)
```

Arguments

target	name of target to make

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, targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE,
  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
verbose	same as for make
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(). Computing this 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 outside of `make()` or `make()`. Also, Windows users will need to download and intall 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

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

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 freeform 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 *Function* 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, targets = drake::possible_targets(plan),
  envir = parent.frame(), verbose = TRUE, jobs = 1,
  parallelism = drake::default_parallelism(), packages = (.packages()),
  prework = character(0), file = character(0), selfcontained = FALSE,
  targets_only = FALSE, config = NULL, font_size = 20,
  layout = "layout_with_sugiyama", direction = "LR",
  navigationButtons = TRUE, hover = TRUE, ...)
```

Arguments

plan	workflow plan data frame, same as for function make().
targets	names of targets to bulid, same as for function make().
envir	environment to import from, same as for function make().
verbose	logical, whether to output messages to the console.
jobs	The outdated() function is called internally, and it needs to import objects and examine your input files to see what has been updated. This could take some time, and parallel computing may be needed to speed up the process. The jobs argument is number of parallel jobs to use for faster computation.
parallelism	Choice of parallel backend to speed up the computation. See ?parallelism_choices for details. The Makefile option is not available here. Drake will try to pick the best option for your system by default.

packages	same as for make().
prework	same as for make().
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.
targets_only	logical, whether to skip the imports and only show the targets in the workflow plan.
config	option internal runtime parameter list of make(...), produced with config(). Computing this in advance could save time if you plan multiple calls to outdated().
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.
direction	an argument to visNetwork::visHierarchicalLayout() indicating the direction of the graph. Options include "LR", "RL", "DU", and "UD". At the time of writing this, the letters must be capitalized, but this may not always be the case ;) in the future.
navigationButtons	logical, whether to add navigation buttons with visNetwork::visInteraction(navigationButtons =
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.
...	other arguments passed to visNetwork::visNetwork() to plot the graph.

See Also

build_graph

Examples

```
## Not run:
load_basic_example()
plot_graph(my_plan, width = "100%") # The width is passed to visNetwork().
make(my_plan)
plot_graph(my_plan) # The red nodes from before are now green.

## 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)
```

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)
```

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)
```

Arguments

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

Value

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)
```

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)
```

---

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 = NULL)
```

Arguments

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

Value

drake target item from the cache

See Also

loadd, 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)
```

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.

Value

a named list of configuration items

See Also

make

Examples

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

## End(Not run)
```

read_graph *Function* read_graph

Description

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

Usage

```
read_graph(path = getwd(), search = TRUE, ...)
```

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.
...	arguments to visNetwork() via plot_graph()

Value

either a plot or an igraph object, depending on plot

See Also

plot_graph, read_config

Examples

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

## End(Not run)
```

read_plan *Function* read_plan

Description

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

Usage

```
read_plan(path = getwd(), search = TRUE)
```

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.

Value

a workflow plan data frame

See Also

read_config

Examples

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

## End(Not run)
```

render_graph *Function* render_graph

Description

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

Usage

```
render_graph(graph, file = character(0), layout = "layout_with_sugiyama",
  direction = "LR", navigationButtons = TRUE, hover = TRUE,
  selfcontained = FALSE, ...)
```

Arguments

graph	list of data frames generated by dataframes_graph(). There should be 3 data frames: nodes, edges, and legend_nodes.
file	Name of HTML file to save the graph. If NULL or character(0), no file is saved and the graph is rendered and displayed within R.
layout	name of an igraph layout to use, such as "layout_with_sugiyama" or "layout_as_tree". Be careful with "layout_as_tree": the graph is a directed acyclic graph, but not necessarily a tree.
direction	an argument to visNetwork::visHierarchicalLayout() indicating the direction of the graph. Options include "LR", "RL", "DU", and "UD". At the time of writing this, the letters must be capitalized, but this may not always be the case ;) in the future.
navigationButtons	logical, whether to add navigation buttons with visNetwork::visInteraction(navigationButtons =
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.
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.
...	arguments passed to visNetwork().

Examples

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

## End(Not run)
```

---

session *Function* session

Description

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

Usage

```
session(path = getwd(), search = TRUE)
```

Arguments

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

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"))
```

Arguments

path	file path of the shell file

See Also

make, max_useful_jobs, parallelism_choices

`status` *Deprecated function* `status`

Description

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

Usage

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

Arguments

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

Value

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

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

See Also

`progress`, `built`, `imported`, `readd`, `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 *Function* summaries

Description

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

Usage

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

Arguments

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

Value

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

See Also

analyses, make, 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 = coef(..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"))
```

---

tracked *Function* tracked

Description

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

Usage

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

Arguments

plan	workflow plan data frame, same as for function make().
targets	names of targets to bulid, same as for function make().
envir	environment to import from, same as for function make().

Examples

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

## End(Not run)
```

Index