Package 'drake'

March 10, 2019

Title A Pipeline Toolkit for Reproducible Computation at Scale

Description A general-purpose computational engine for data analysis,
drake rebuilds intermediate data objects when their dependencies change,
and it skips work when the results are already up to date.
Not every execution starts from scratch,
there is native support for parallel and distributed computing,
and completed projects have tangible evidence
that they are reproducible.
Extensive documentation, from beginner-friendly tutorials
to practical examples and more, is available at the reference website
<https://ropensci.github.io/drake/> and the online manual
<https://ropenscilabs.github.io/drake-manual/>.

Version 7.0.0

License GPL-3

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

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

Depends R (>= 3.3.0)

Imports base64url, digest, igraph, methods, rlang (>= 0.2.0), storr
(>= 1.1.0), utils

Suggests abind, bindr, callr, clustermq, CodeDepends, crayon, curl,
datasets, downloader, future, grDevices, ggplot2, ggraph,
knitr, lubridate, networkD3, parallel, prettycode, stats,
testthat, tibble, tidyselect (>= 0.2.4), rmarkdown, styler,
visNetwork, webshot

VignetteBuilder knitr

Encoding UTF-8

Language en-US

RoxygenNote 6.1.1

NeedsCompilation yes

Author William Michael Landau [aut, cre]
(<https://orcid.org/0000-0003-1878-3253>),
Alex Axthelm [ctb],
Jasper Clarkberg [ctb],
Kirill Müller [ctb],
Ben Marwick [rev],
Peter Slaughter [rev],
Ben Bond-Lamberty [ctb] (<https://orcid.org/0000-0001-9525-4633>),
Tristan Mahr [ctb] (<https://orcid.org/0000-0002-8890-5116>),
Eli Lilly and Company [cph]

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

Repository CRAN

Date/Publication 2019-03-10 21:33:10 UTC

R topics documented:

`drake-package` *drake: A pipeline toolkit for reproducible computation at scale.*

Description

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

Author(s)

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

References

https://github.com/ropensci/drake

Examples

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

## End(Not run)
```

---

bind_plans *Row-bind together drake plans*

Description

Combine drake plans together in a way that correctly fills in missing entries.

Usage

```
bind_plans(...)
```

Arguments

...	Workflow plan data frames (see drake_plan()).

See Also

drake_plan(), make()

Examples

```
# You might need to refresh your data regularly (see ?triggers).
download_plan <- drake_plan(
  data = target(
    command = download_data(),
    trigger = "always"
  )
)
# But if the data don't change, the analyses don't need to change.
analysis_plan <- drake_plan(
  usage = get_usage_metrics(data),
  topline = scrape_topline_table(data)
)
your_plan <- bind_plans(download_plan, analysis_plan)
your_plan
# make(your_plan) # nolint
```

---

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

Description

Applies to targets in your plan, not imports or files.

Usage

```
build_times(..., path = getwd(), search = TRUE, digits = 3,
  cache = get_cache(path = path, search = search, verbose = verbose),
  targets_only = NULL, verbose = 1L, jobs = 1, type = c("build",
  "command"))
```

Arguments

...	Targets to load from the cache: as names (symbols) or character strings. If the tidyselect package is installed, you can also supply dplyr-style tidyselect commands such as starts_with(), ends_with(), and one_of().
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project. Ignored if a cache is supplied.
search	Logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a cache is supplied.
digits	How many digits to round the times to.
cache	drake cache. See new_cache(). If supplied, path and search are ignored.
targets_only	Deprecated.
verbose	Logical or numeric, control printing to the console. • 0 or FALSE: print nothing. • 1 or TRUE: print only targets to build. • 2: plus checks and cache info. • 3: plus missing imports. • 4: plus all imports. • 5: plus execution and total build times for targets. • 6: plus notifications when targets are being stored.
jobs	Number of jobs/workers for parallel processing.
type	Type of time you want: either "build" for the full build time including the time it took to store the target, or "command" for the time it took just to run the command.

Value

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

See Also

predict_runtime()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
if (requireNamespace("lubridate")) {
# Show the build times for the mtcars example.
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Build all the targets.
print(build_times()) # Show how long it took to build each target.
}
}
})

## End(Not run)
```

cached *List targets in the cache.*

Description

Tip: read/load a cached item with `readd()` or `loadd()`.

Usage

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

Arguments

`...`	Deprecated. Do not use. Objects to load from the cache, as names (unquoted) or character strings (quoted). Similar to `...` in `remove()`.
`list`	Deprecated. Do not use. Character vector naming objects to be loaded from the cache. Similar to the `list` argument of `remove()`.
`no_imported_objects`	Logical, deprecated. Use `targets_only` instead.
`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project. Ignored if a `cache` is supplied.
`search`	Logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a `cache` is supplied.
`cache`	drake cache. See `new_cache()`. If supplied, `path` and `search` are ignored.
`verbose`	Logical or numeric, control printing to the console. • `0` or `FALSE`: print nothing. • `1` or `TRUE`: print only targets to build. • `2`: plus checks and cache info. • `3`: plus missing imports. • `4`: plus all imports. • `5`: plus execution and total build times for targets. • `6`: plus notifications when targets are being stored.
`namespace`	Character scalar, name of the storr namespace to use for listing objects.
`jobs`	Number of jobs/workers for parallel processing.
`targets_only`	Logical. If `TRUE` just list the targets. If `FALSE`, list files and imported objects too.

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

readd(), loadd(), drake_plan(), make()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
if (requireNamespace("lubridate")) {
load_mtcars_example() # Load drake's canonical example.
make(my_plan) # Run the project, build all the targets.
cached()
cached(targets_only = FALSE)
}
}
})

## End(Not run)
```

---

clean *Remove targets/imports from the cache.*

Description

Cleans up the work done by make().

Usage

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

Arguments

...	Targets to remove from the cache: as names (symbols) or character strings. If the tidyselect package is installed, you can also supply dplyr-style tidyselect commands such as starts_with(), ends_with(), and one_of().
list	Character vector naming targets to be removed from the cache. Similar to the list argument of remove().
destroy	Logical, whether to totally remove the drake cache. If destroy is FALSE, only the targets from make() are removed. If TRUE, the whole cache is removed, including session metadata, etc.
path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project. Ignored if a cache is supplied.
search	Logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a cache is supplied.

cache — drake cache. See new_cache(). If supplied, path and search are ignored.

verbose — Logical or numeric, control printing to the console.

- 0 or FALSE: print nothing.
- 1 or TRUE: print only targets to build.
- 2: plus checks and cache info.
- 3: plus missing imports.
- 4: plus all imports.
- 5: plus execution and total build times for targets.
- 6: plus notifications when targets are being stored.

jobs — Number of jobs for light parallelism (disabled on Windows).

force — Logical, whether to try to clean the cache even though the project may not be back compatible with the current version of drake.

garbage_collection — Logical, whether to call cache$gc() to do garbage collection. If TRUE, cached data with no remaining references will be removed. This will slow down clean(), but the cache could take up far less space afterwards. See the gc() method for storr caches.

purge — Logical, whether to remove objects from metadata namespaces such as "meta", "build_times", and "errors".

Details

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

Value

Invisibly return NULL.

See Also

drake_gc(), make()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Run the project, build the targets.
# List objects in the cache, excluding R objects
```

```
# imported from your workspace.
cached(no_imported_objects = TRUE)
# Remove 'summ_regression1_large' and 'small' from the cache.
clean(summ_regression1_large, small)
# Those objects should be gone.
cached(no_imported_objects = TRUE)
# Rebuild the missing targets.
make(my_plan)
# Remove all the targets and imports.
# On non-Windows machines, parallelize over at most 2 jobs.
clean(jobs = 2)
# Make the targets again.
make(my_plan)
# Garbage collection removes data whose references are no longer present.
# It is slow, but you should enable it if you want to reduce the
# size of the cache.
clean(garbage_collection = TRUE)
# All the targets and imports are gone.
cached()
# But there is still cached metadata.
build_times()
# To make even more room, use the "purge" flag.
clean(purge = TRUE)
build_times()
# Completely remove the entire cache (default: '.drake/' folder).
clean(destroy = TRUE)
}
})

## End(Not run)
```

`clean_mtcars_example` *Clean the mtcars example from* `drake_example("mtcars")`

Description

This function deletes files. Use at your own risk. Destroys the `.drake/` cache and the `report.Rmd` file in the current working directory. Your working directory (`getcwd()`) must be the folder from which you first ran `load_mtcars_example()` and `make(my_plan)`.

Usage

```
clean_mtcars_example()
```

Value

nothing

See Also

`load_mtcars_example()`, `clean()`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
# Populate your workspace and write 'report.Rmd'.
load_mtcars_example() # Get the code: drake_example("mtcars")
# Check the dependencies of an imported function.
deps_code(reg1)
# Check the dependencies of commands in the workflow plan.
deps_code(my_plan$command[1])
deps_code(my_plan$command[4])
# Plot the interactive network visualization of the workflow.
config <- drake_config(my_plan)
outdated(config) # Which targets are out of date?
# Run the workflow to build all the targets in the plan.
make(my_plan)
outdated(config) # Everything should be up to date.
# For the reg2() model on the small dataset,
# the p-value is so small that there may be an association
# between weight and fuel efficiency after all.
readd(coef_regression2_small)
# Clean up the example.
clean_mtcars_example()
}
})

## End(Not run)
```

---

code_to_plan *Turn an R script file or knitr / R Markdown report into a* drake *workflow plan data frame.*

Description

code_to_plan(), plan_to_code(), and plan_to_notebook() together illustrate the relationships between drake plans, R scripts, and R Markdown documents.

Usage

```
code_to_plan(path)
```

Arguments

path	A file path to an R script or knitr report.

Details

This feature is easy to break, so there are some rules for your code file:

1. Stick to assigning a single expression to a single target at a time. For multi-line commands, please enclose the whole command in curly braces. Conversely, compound assignment is not supported (e.g. `target_1 <- target_2 <- target_3 <- get_data()`).
2. Once you assign an expression to a variable, do not modify the variable any more. The target/command binding should be permanent.
3. Keep it simple. Please use the assignment operators rather than `assign()` and similar functions.

See Also

`drake_plan()`, `make()`, `plan_to_code()`, `plan_to_notebook()`

Examples

```
plan <- drake_plan(
  raw_data = read_excel(file_in("raw_data.xlsx")),
  data = raw_data,
  hist = create_plot(data),
  fit = lm(Sepal.Width ~ Petal.Width + Species, data)
)
file <- tempfile()
# Turn the plan into an R script a the given file path.
plan_to_code(plan, file)
# Here is what the script looks like.
cat(readLines(file), sep = "\n")
# Convert back to a drake plan.
if (requireNamespace("CodeDepends")) {
  code_to_plan(file)
}
```

---

deps_code *List the dependencies of a function or command*

Description

Functions are assumed to be imported, and language/text are assumed to be commands in a plan.

Usage

```
deps_code(x)
```

Arguments

x	A function, expression, or text.

Value

A data frame of the dependencies.

See Also

deps_target(), deps_knitr()

Examples

```
# Your workflow likely depends on functions in your workspace.
f <- function(x, y) {
  out <- x + y + g(x)
  saveRDS(out, "out.rds")
}
# Find the dependencies of f. These could be R objects/functions
# in your workspace or packages. Any file names or target names
# will be ignored.
deps_code(f)
# Define a workflow plan data frame that uses your function f().
my_plan <- drake_plan(
  x = 1 + some_object,
  my_target = x + readRDS(file_in("tracked_input_file.rds")),
  return_value = f(x, y, g(z + w))
)
# Get the dependencies of workflow plan commands.
# Here, the dependencies could be R functions/objects from your workspace
# or packages, imported files, or other targets in the workflow plan.
deps_code(my_plan$command[[1]])
deps_code(my_plan$command[[2]])
deps_code(my_plan$command[[3]])
# You can also supply expressions or text.
deps_code(quote(x + y + 123))
deps_code("x + y + 123")
```

---

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

Description

Dependencies in knitr reports are marked by loadd() and readd() in active code chunks.

Usage

```
deps_knitr(path)
```

Arguments

path	Encoded file path to the knitr/R Markdown document. Wrap paths in file_store() to encode.

Value

A data frame of dependencies.

See Also

deps_code(), deps_target()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
deps_knitr("report.Rmd")
})

## End(Not run)
```

---

deps_profile *Find out why a target is out of date.*

Description

The dependency profile can give you a hint as to why a target is out of date. It can tell you if

- at least one input file changed,
- at least one output file changed,
- or a non-file dependency changed. For this last part, the imports need to be up to date in the cache, which you can do with `outdated()` or `make(skip_targets = TRUE)`. Unfortunately, `deps_profile()` does not currently get more specific than that.

Usage

```
deps_profile(target, config, character_only = FALSE)
```

Arguments

target	Name of the target.
config	Configuration list output by drake_config() or make().
character_only	Logical, whether to assume target is a character string rather than a symbol.

Value

A data frame of the old hashes and new hashes of the data frame, along with an indication of which hashes changed since the last make().

See Also

diagnose(), deps_code(), make(), drake_config()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Load drake's canonical example.
make(my_plan) # Run the project, build the targets.
config <- drake_config(my_plan)
# Get some example dependency profiles of targets.
deps_profile(small, config = config)
# Change a dependency.
simulate <- function(x) {}
# Update the in-memory imports in the cache
# so deps_profile can detect changes to them.
# Changes to targets are already cached.
make(my_plan, skip_targets = TRUE)
# The dependency hash changed.
deps_profile(small, config = config)
}
})

## End(Not run)
```

---

deps_target *List the dependencies of a target*

Description

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

Usage

```
deps_target(target, config, character_only = FALSE)
```

Arguments

target	A symbol denoting a target name, or if character_only is TRUE, a character scalar denoting a target name.
config	An output list from drake_config().
character_only	Logical, whether to assume target is a character string rather than a symbol.

Value

A data frame with the dependencies listed by type (globals, files, etc).

See Also

deps_code(), deps_knitr()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
config <- drake_config(my_plan)
deps_target("regression1_small", config = config)
})

## End(Not run)
```

`diagnose` *Get diagnostic metadata on a target.*

Description

Diagnostics include errors, warnings, messages, runtimes, and other context/metadata from when a target was built or an import was processed. If your target's last build succeeded, then `diagnose(your_target)` has the most current information from that build. But if your target failed, then only `diagnose(your_target)$error`, `diagnose(your_target)$warnings`, and `diagnose(your_target)$messages` correspond to the failure, and all the other metadata correspond to the last build that completed without an error.

Usage

```
diagnose(target = NULL, character_only = FALSE, path = getwd(),
  search = TRUE, cache = drake::get_cache(path = path, search = search,
  verbose = verbose), verbose = 1L)
```

Arguments

`target`	Name of the target of the error to get. Can be a symbol if `character_only` is `FALSE`, must be a character if `character_only` is `TRUE`.
`character_only`	Logical, whether `target` should be treated as a character or a symbol. Just like `character.only` in `library()`.
`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project. Ignored if a `cache` is supplied.
`search`	Logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a `cache` is supplied.
`cache`	drake cache. See `new_cache()`. If supplied, `path` and `search` are ignored.
`verbose`	Logical or numeric, control printing to the console. • `0` or `FALSE`: print nothing. • `1` or `TRUE`: print only targets to build. • `2`: plus checks and cache info. • `3`: plus missing imports. • `4`: plus all imports. • `5`: plus execution and total build times for targets. • `6`: plus notifications when targets are being stored.

Value

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

See Also

failed(), progress(), readd(), drake_plan(), make()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
diagnose() # List all the targets with recorded error logs.
# Define a function doomed to failure.
f <- function() {
  stop("unusual error")
}
# Create a workflow plan doomed to failure.
bad_plan <- drake_plan(my_target = f())
# Running the project should generate an error
# when trying to build 'my_target'.
try(make(bad_plan), silent = FALSE)
failed() # List the failed targets from the last make() (my_target).
# List targets that failed at one point or another
# over the course of the project (my_target).
# drake keeps all the error logs.
diagnose()
# Get the error log, an object of class "error".
error <- diagnose(my_target)$error # See also warnings and messages.
str(error) # See what's inside the error log.
error$calls # View the traceback. (See the traceback() function).
suppressWarnings(
  make(
    drake_plan(
      x = 1,
      y = warning(123),
      z = warning(456)
    ),
    verbose = FALSE
  )
)
targets <- built(verbose = FALSE)
out <- lapply(targets, diagnose, character_only = TRUE, verbose = FALSE)
names(out) <- targets
unlist(out)
})

## End(Not run)
```

drake_build *Build/process a single target or import.*

Description

Also load the target's dependencies beforehand.

Usage

```
drake_build(target, config = NULL, meta = NULL,
  character_only = FALSE, envir = NULL, jobs = 1, replace = FALSE)
```

Arguments

target	Name of the target.
config	Internal configuration list.
meta	Deprecated.
character_only	Logical, whether name should be treated as a character or a symbol (just like character.only in library()).
envir	Environment to load objects into. Defaults to the calling environment (current workspace).
jobs	Number of parallel jobs for loading objects. On non-Windows systems, the loading process for multiple objects can be lightly parallelized via parallel::mclapply(). just set jobs to be an integer greater than 1. On Windows, jobs is automatically demoted to 1.
replace	Logical. If FALSE, items already in your environment will not be replaced.

Value

The value of the target right after it is built.

See Also

drake_debug()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
# This example is not really a user-side demonstration.
# It just walks through a dive into the internals.
# Populate your workspace and write 'report.Rmd'.
load_mtcars_example() # Get the code with drake_example("mtcars").
# Create the master internal configuration list.
config <- drake_config(my_plan)
```

```
out <- drake_build(small, config = config)
# Now includes `small`.
cached()
head(readd(small))
# `small` was invisibly returned.
head(out)
# If you previously called make(),
# `config` is just read from the cache.
make(my_plan, verbose = FALSE)
config <- drake_config(my_plan)
result <- drake_build(small, config = config)
head(result)
}
})

## End(Not run)
```

---

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

Description

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

Usage

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

Arguments

path	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project. Ignored if a `cache` is supplied.
search	Logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a `cache` is supplied.
cache	drake cache. See `new_cache()`. If supplied, `path` and `search` are ignored.
verbose	Logical or numeric, control printing to the console. • `0` or `FALSE`: print nothing. • `1` or `TRUE`: print only targets to build. • `2`: plus checks and cache info. • `3`: plus missing imports. • `4`: plus all imports. • `5`: plus execution and total build times for targets. • `6`: plus notifications when targets are being stored.

jobs	Number of jobs/workers for parallel processing.
targets_only	Logical, whether to output information only on the targets in your workflow plan data frame. If targets_only is FALSE, the output will include the hashes of both targets and imports.

Details

A hash is a fingerprint of an object's value. Together, the hash keys of all your targets and imports represent the state of your project. Use drake_cache_log() to generate a data frame with the hash keys of all the targets and imports stored in your cache. This function is particularly useful if you are storing your drake project in a version control repository. The cache has a lot of tiny files, so you should not put it under version control. Instead, save the output of drake_cache_log() as a text file after each make(), and put the text file under version control. That way, you have a changelog of your project's results. See the examples below for details. Depending on your project's history, the targets may be different than the ones in your workflow plan data frame. Also, the keys depend on the hash algorithm of your cache. To define your own hash algorithm, you can create your own storr cache and give it a hash algorithm (e.g. storr_rds(hash_algorithm = "murmur32"))

Value

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

See Also

cached(), get_cache()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
# Load drake's canonical example.
load_mtcars_example() # Get the code with drake_example()
# Run the project, build all the targets.
make(my_plan)
# Get a data frame of all the hash keys.
# If you want a changelog, be sure to do this after every make().
cache_log <- drake_cache_log()
head(cache_log)
# Suppress partial arg match warnings.
suppressWarnings(
  # Save the hash log as a flat text file.
  write.table(
    x = cache_log,
    file = "drake_cache.log",
    quote = FALSE,
    row.names = FALSE
  )
)
# At this point, put drake_cache.log under version control
# (e.g. with 'git add drake_cache.log') alongside your code.
```

```
# Now, every time you run your project, your commit history
# of hash_lot.txt is a changelog of the project's results.
# It shows which targets and imports changed on every commit.
# It is extremely difficult to track your results this way
# by putting the raw '.drake/' cache itself under version control.
}
})

## End(Not run)
```

---

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

Description

drake_config() collects and sanitizes the multitude of parameters and settings that make() needs to do its job: the plan, packages, the environment of functions and initial data objects, parallel computing instructions, verbosity level, etc. Other functions such as outdated(), vis_drake_graph(), and predict_runtime() require output from drake_config() for the config argument. If you supply a drake_config() object to the config argument of make(), then drake will ignore all the other arguments because it already has everything it needs in config.

Usage

```
drake_config(plan, targets = NULL, envir = parent.frame(),
  verbose = 1L, hook = NULL, cache = drake::get_cache(verbose =
  verbose, console_log_file = console_log_file), fetch_cache = NULL,
  parallelism = "loop", jobs = 1L, jobs_preprocess = 1L,
  packages = rev(.packages()), lib_loc = NULL,
  prework = character(0), prepend = NULL, command = NULL,
  args = NULL, recipe_command = NULL, timeout = NULL, cpu = Inf,
  elapsed = Inf, retries = 0, force = FALSE, log_progress = FALSE,
  graph = NULL, trigger = drake::trigger(), skip_targets = FALSE,
  skip_imports = FALSE, skip_safety_checks = FALSE,
  lazy_load = "eager", session_info = TRUE, cache_log_file = NULL,
  seed = NULL, caching = c("master", "worker"), keep_going = FALSE,
  session = NULL, pruning_strategy = NULL, makefile_path = NULL,
  console_log_file = NULL, ensure_workers = TRUE,
  garbage_collection = FALSE, template = list(), sleep = function(i)
  0.01, hasty_build = NULL, memory_strategy = c("speed", "memory",
  "lookahead"), layout = NULL, lock_envir = TRUE)
```

Arguments

plan	Workflow plan data frame. A workflow plan data frame is a data frame with a target column and a command column. (See the details in the drake_plan() help file for descriptions of the optional columns.) Targets are the objects that drake generates, and commands are the pieces of R code that produce them. You

can create and track custom files along the way (see `file_in()`, `file_out()`, and `knitr_in()`). Use the function `drake_plan()` to generate workflow plan data frames.

`targets` Character vector, names of targets to build. Dependencies are built too. Together, the `plan` and `targets` comprise the workflow network (i.e. the `graph` argument). Changing either will change the network.

`envir` Environment to use. Defaults to the current workspace, so you should not need to worry about this most of the time. A deep copy of `envir` is made, so you don't need to worry about your workspace being modified by `make`. The deep copy inherits from the global environment. Wherever necessary, objects and functions are imported from `envir` and the global environment and then reproducibly tracked as dependencies.

`verbose` Logical or numeric, control printing to the console.

- `0` or `FALSE`: print nothing.
- `1` or `TRUE`: print only targets to build.
- `2`: plus checks and cache info.
- `3`: plus missing imports.
- `4`: plus all imports.
- `5`: plus execution and total build times for targets.
- `6`: plus notifications when targets are being stored.

`hook` Deprecated.

`cache` drake cache as created by `new_cache()`. See also `get_cache()`.

`fetch_cache` Deprecated.

`parallelism` Character scalar, type of parallelism to use. For detailed explanations, see the high-performance computing chapter of the user manual.

You could also supply your own scheduler function if you want to experiment or aggressively optimize. The function should take a single `config` argument (produced by `drake_config()`). Existing examples from `drake`'s internals are the `backend_*()` functions:

- `backend_loop()`
- `backend_clustermq()`
- `backend_future()`
- `backend_hasty()` (unofficial) However, this functionality is really a back door and should not be used for production purposes unless you really know what you are doing and you are willing to suffer setbacks whenever `drake`'s unexported core functions are updated.

`jobs` Maximum number of parallel workers for processing the targets. You can experiment with `predict_runtime()` to help decide on an appropriate number of jobs. For details, visit `https://ropenscilabs.github.io/drake-manual/time.html`.

`jobs_preprocess`

Number of parallel jobs for processing the imports and doing other preprocessing tasks.

packages — Character vector packages to load, in the order they should be loaded. Defaults to `rev(.packages())`, so you should not usually need to set this manually. Just call `library()` to load your packages before `make()`. However, sometimes packages need to be strictly forced to load in a certain order, especially if `parallelism` is `"Makefile"`. To do this, do not use `library()` or `require()` or `loadNamespace()` or `attachNamespace()` to load any libraries beforehand. Just list your packages in the `packages` argument in the order you want them to be loaded.

lib_loc — Character vector, optional. Same as in `library()` or `require()`. Applies to the `packages` argument (see above).

prework — Expression (language object), list of expressions, or character vector. Code to run right before targets build. Called only once if `parallelism` is `"loop"` and once per target otherwise. This code can be used to set global options, etc.

prepend — Deprecated.

command — Deprecated.

args — Deprecated.

recipe_command — Deprecated.

timeout — deprecated. Use `elapsed` and `cpu` instead.

cpu — Same as the `cpu` argument of `setTimeLimit()`. Seconds of cpu time before a target times out. Assign target-level cpu timeout times with an optional `cpu` column in `plan`.

elapsed — Same as the `elapsed` argument of `setTimeLimit()`. Seconds of elapsed time before a target times out. Assign target-level elapsed timeout times with an optional `elapsed` column in `plan`.

retries — Number of retries to execute if the target fails. Assign target-level retries with an optional `retries` column in `plan`.

force — Logical. If `FALSE` (default) then `drake` imposes checks if the cache was created with an old and incompatible version of drake. If there is an incompatibility, `make()` stops to give you an opportunity to downgrade `drake` to a compatible version rather than rerun all your targets from scratch.

log_progress — Logical, whether to log the progress of individual targets as they are being built. Progress logging creates a lot of little files in the cache, and it may make builds a tiny bit slower. So you may see gains in storage efficiency and speed with `make(..., log_progress = FALSE)`.

graph — An `igraph` object from the previous `make()`. Supplying a pre-built graph could save time.

trigger — Name of the trigger to apply to all targets. Ignored if `plan` has a `trigger` column. See `trigger()` for details.

skip_targets — Logical, whether to skip building the targets in `plan` and just import objects and files.

skip_imports — Logical, whether to totally neglect to process the imports and jump straight to the targets. This can be useful if your imports are massive and you just want to test your project, but it is bad practice for reproducible data analysis. This argument is overridden if you supply your own `graph` argument.

skip_safety_checks
: Logical, whether to skip the safety checks on your workflow. Use at your own peril.

lazy_load
: Either a character vector or a logical. Choices:

  - "eager": no lazy loading. The target is loaded right away with assign().
  - "promise": lazy loading with delayedAssign()
  - "bind": lazy loading with active bindings: bindr::populate_env().
  - TRUE: same as "promise".
  - FALSE: same as "eager".

  lazy_load should not be "promise" for "parLapply" parallelism combined with jobs greater than 1. For local multi-session parallelism and lazy loading, try library(future); future::plan(multisession) and then make(..., parallelism = "future_

  If lazy_load is "eager", drake prunes the execution environment before each target/stage, removing all superfluous targets and then loading any dependencies it will need for building. In other words, drake prepares the environment in advance and tries to be memory efficient. If lazy_load is "bind" or "promise", drake assigns promises to load any dependencies at the last minute. Lazy loading may be more memory efficient in some use cases, but it may duplicate the loading of dependencies, costing time.

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

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

seed
: Integer, the root pseudo-random number generator seed to use for your project. In make(), drake generates a unique local seed for each target using the global seed and the target name. That way, different pseudo-random numbers are generated for different targets, and this pseudo-randomness is reproducible.

  To ensure reproducibility across different R sessions, set.seed() and .Random.seed are ignored and have no affect on drake workflows. Conversely, make() does not usually change .Random.seed, even when pseudo-random numbers are generated. The exceptions to this last point are make(parallelism = "clustermq") and make(parallelism = "clustermq_staged"), because the clustermq package needs to generate random numbers to set up ports and sockets for ZeroMQ.

  On the first call to make() or drake_config(), drake uses the random number generator seed from the seed argument. Here, if the seed is NULL (default), drake uses a seed of 0. On subsequent make()s for existing projects, the project's cached seed will be used in order to ensure reproducibility. Thus, the seed argument must either be NULL or the same seed from the project's cache

(usually the .drake/ folder). To reset the random number generator seed for a project, use clean(destroy = TRUE).

caching Character string, only applies to "clustermq", "clustermq_staged", and "future" parallel backends. The caching argument can be either "master" or "worker".

- "master": Targets are built by remote workers and sent back to the master process. Then, the master process saves them to the cache (config$cache, usually a file system storr). Appropriate if remote workers do not have access to the file system of the calling R session. Targets are cached one at a time, which may be slow in some situations.
- "worker": Remote workers not only build the targets, but also save them to the cache. Here, caching happens in parallel. However, remote workers need to have access to the file system of the calling R session. Transferring target data across a network can be slow.

keep_going Logical, whether to still keep running make() if targets fail.

session Deprecated. Has no effect now.

pruning_strategy
Deprecated. See memory_strategy.

makefile_path Path to the Makefile for make(parallelism = "Makefile"). If you set this argument to a non-default value, you are responsible for supplying this same path to the args argument so make knows where to find it. Example: make(parallelism = "Makefile", makefile_path = ".drake/.makefile", command = "make", a

console_log_file
Character scalar, connection object (such as stdout()) or NULL. If NULL, console output will be printed to the R console using message(). If a character scalar, console_log_file should be the name of a flat file, and console output will be appended to that file. If a connection object (e.g. stdout()) warnings and messages will be sent to the connection. For example, if console_log_file is stdout(), warnings and messages are printed to the console in real time (in addition to the usual in-bulk printing after each target finishes).

ensure_workers Logical, whether the master process should wait for the workers to post before assigning them targets. Should usually be TRUE. Set to FALSE for make(parallelism = "future_lapply (n > 1) when combined with future::plan(future::sequential). This argument only applies to parallel computing with persistent workers (make(parallelism = x), where x could be "mclapply", "parLapply", or "future_lapply").

garbage_collection
Logical, whether to call gc() each time a target is built during make().

template A named list of values to fill in the {{ ... }} placeholders in template files (e.g. from drake_hpc_template_file()). Same as the template argument of clustermq::Q() and clustermq::workers. Enabled for clustermq only (make(parallelism = "clustermq_staged")), not future or batchtools so far. For more information, see the clustermq package: https://github.com/mschubert/clustermq. Some template placeholders such as {{ job_name }} and {{ n_jobs }} cannot be set this way.

sleep In its parallel processing, drake uses a central master process to check what the parallel workers are doing, and for the affected high-performance computing

workflows, wait for data to arrive over a network. In between loop iterations, the master process sleeps to avoid throttling. The `sleep` argument to `make()` and `drake_config()` allows you to customize how much time the master process spends sleeping.

The `sleep` argument is a function that takes an argument `i` and returns a numeric scalar, the number of seconds to supply to `Sys.sleep()` after iteration `i` of checking. (Here, `i` starts at 1.) If the checking loop does something other than sleeping on iteration `i`, then `i` is reset back to 1.

To sleep for the same amount of time between checks, you might supply something like `function(i) 0.01`. But to avoid consuming too many resources during heavier and longer workflows, you might use an exponential back-off: say, `function(i) { 0.1 + 120 * pexp(i - 1, rate = 0.01) }`.

`hasty_build` A user-defined function. In "hasty mode" (`make(parallelism = "hasty")`) this is the function that evaluates a target's command and returns the resulting value. The `hasty_build` argument has no effect if `parallelism` is any value other than "hasty".

The function you pass to `hasty_build` must have arguments `target` and `config`. Here, `target` is a character scalar naming the target being built, and `config` is a configuration list of runtime parameters generated by `drake_config()`.

`memory_strategy`

Character scalar, name of the strategy `drake` uses to manage targets in memory. For more direct control over which targets `drake` keeps in memory, see the help file examples of `drake_envir()`. The `memory_strategy` argument to `make()` and `drake_config()` is an attempt at an automatic catch-all solution. These are the choices.

- `"speed"`: Once a target is loaded in memory, just keep it there. This choice maximizes speed and hogs memory.
- `"memory"`: Just before building each new target, unload everything from memory except the target's direct dependencies. This option conserves memory, but it sacrifices speed because each new target needs to reload any previously unloaded targets from storage.
- `"lookahead"`: Just before building each new target, search the dependency graph to find targets that will not be needed for the rest of the current `make()` session. In this mode, targets are only in memory if they need to be loaded, and we avoid superfluous reads from the cache. However, searching the graph takes time, and it could even double the computational overhead for large projects.

Each strategy has a weakness. `"speed"` is memory-hungry, `"memory"` wastes time reloading targets from storage, and `"lookahead"` wastes time traversing the entire dependency graph on every `make()`. For a better compromise and more control, see the examples in the help file of `drake_envir()`.

`layout` `config$layout`, where `config` is the return value from a prior call to `drake_config()`. If your plan or environment have changed since the last `make()`, do not supply a `layout` argument. Otherwise, supplying one could save time.

`lock_envir` Logical, whether to lock `config$envir` during `make()`. If `TRUE`, `make()` quits in error whenever a command in your `drake` plan (or `prework`) tries to add,

remove, or modify non-hidden variables in your environment/workspace/R session. This is extremely important for ensuring the purity of your functions and the reproducibility/credibility/trust you can place in your project. lock_envir will be set to a default of TRUE in drake version 7.0.0 and higher.

Value

The master internal configuration list of a project.

See Also

make(), drake_plan(), vis_drake_graph()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
# Construct the master internal configuration list.
config <- drake_config(my_plan)
if (requireNamespace("visNetwork")) {
  vis_drake_graph(config) # See the dependency graph.
  if (requireNamespace("networkD3")) {
    sankey_drake_graph(config) # See the dependency graph.
  }
}
# These functions are faster than otherwise
# because they use the configuration list.
outdated(config) # Which targets are out of date?
missed(config) # Which imports are missing?
})

## End(Not run)
```

---

drake_debug *Run a single target's command in debug mode.*

Description

drake_debug() loads a target's dependencies and then runs its command in debug mode (see browser(), debug(), and debugonce()). This function does not store the target's value in the cache (see https://github.com/ropensci/drake/issues/587).

Usage

```
drake_debug(target = NULL, config = NULL, character_only = FALSE,
  envir = NULL, jobs = 1, replace = FALSE, verbose = TRUE)
```

Arguments

target	Name of the target.
config	Internal configuration list.
character_only	Logical, whether name should be treated as a character or a symbol (just like character.only in library()).
envir	Environment to load objects into. Defaults to the calling environment (current workspace).
jobs	Number of parallel jobs for loading objects. On non-Windows systems, the loading process for multiple objects can be lightly parallelized via parallel::mclapply(). just set jobs to be an integer greater than 1. On Windows, jobs is automatically demoted to 1.
replace	Logical. If FALSE, items already in your environment will not be replaced.
verbose	Logical, whether to print out the target you are debugging.

Value

The value of the target right after it is built.

See Also

drake_build()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
# This example is not really a user-side demonstration.
# It just walks through a dive into the internals.
# Populate your workspace and write 'report.Rmd'.
load_mtcars_example() # Get the code with drake_example("mtcars").
# Create the master internal configuration list.
config <- drake_config(my_plan)
out <- drake_build(small, config = config)
# Now includes `small`.
cached()
head(readd(small))
# `small` was invisibly returned.
head(out)
# If you previously called make(),
# `config` is just read from the cache.
make(my_plan, verbose = FALSE)
result <- drake_build(small, config = config)
head(result)
}
})

## End(Not run)
```

drake_envir *Get the environment where drake builds targets*

Description

Call this function inside the commands in your plan to get the environment where drake builds targets. That way, you can strategically remove targets from memory while `make()` is running. That way, you can limit the amount of computer memory you use.

Usage

```
drake_envir()
```

Value

The environment where drake builds targets.

See Also

`from_plan()`

Examples

```
plan <- drake_plan(
  large_data_1 = sample.int(1e4),
  large_data_2 = sample.int(1e4),
  subset = c(large_data_1[seq_len(10)], large_data_2[seq_len(10)]),
  summary = {
    print(ls(envir = drake_envir()))
    # We don't need the large_data_* targets in memory anymore.
    rm(large_data_1, large_data_2, envir = drake_envir())
    print(ls(envir = drake_envir()))
    mean(subset)
  }
)
make(plan, cache = storr::storr_environment(), session_info = FALSE)
```

drake_example *Download and save the code and data files of an example* drake*-powered project.*

Description

The `drake_example()` function downloads a folder from https://github.com/wlandau/drake-examples. (Really, it downloads one of the zip files listed at https://github.com/wlandau/drake-examples/tree/gh-pages and unzips it. Do not include the `.zip` extension in the `example` argument.)

Usage

```
drake_example(example = "main", to = getwd(), destination = NULL,
  overwrite = FALSE, quiet = TRUE)
```

Arguments

example	Name of the example. The possible values are the names of the folders at https://github.com/wlandau/drake-examples.
to	Character scalar, the folder containing the code files for the example. passed to the exdir argument of utils::unzip().
destination	Deprecated; use to instead.
overwrite	Logical, whether to overwrite an existing folder with the same name as the drake example.
quiet	Logical, passed to downloader::download() and thus utils::download.file(). Whether to download quietly or print progress.

Value

NULL

See Also

drake_examples(), make()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (requireNamespace("downloader")) {
drake_examples() # List all the drake examples.
# Sets up the same example as https://ropenscilabs.github.io/drake-manual/mtcars.html # nolint
drake_example("mtcars")
# Sets up the SLURM example.
drake_example("slurm")
}
})

## End(Not run)
```

---

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

Description

You can find the code files of the examples at https://github.com/wlandau/drake-examples. The drake_examples() function downloads the list of examples from https://wlandau.github.io/drake-examples/examples.md, so you need an internet connection.

Usage

```
drake_examples(quiet = TRUE)
```

Arguments

quiet	Logical, passed to `downloader::download()` and thus `utils::download.file()`. Whether to download quietly or print progress.

Value

Names of all the drake examples.

See Also

`drake_example()`, `make()`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (requireNamespace("downloader")) {
drake_examples() # List all the drake examples.
# Sets up the example from
# https://ropenscilabs.github.io/drake-manual/mtcars.html
drake_example("mtcars")
# Sets up the SLURM example.
drake_example("slurm")
}
})

## End(Not run)
```

---

drake_gc *Do garbage collection on the drake cache.*

Description

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

Usage

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

Arguments

path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project. Ignored if a cache is supplied.
search	Logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a cache is supplied.
verbose	Logical or numeric, control printing to the console. • 0 or FALSE: print nothing. • 1 or TRUE: print only targets to build. • 2: plus checks and cache info. • 3: plus missing imports. • 4: plus all imports. • 5: plus execution and total build times for targets. • 6: plus notifications when targets are being stored.
cache	drake cache. See new_cache(). If supplied, path and search are ignored.
force	Logical, whether to load the cache despite any back compatibility issues with the running version of drake.

Value

NULL

See Also

clean()

Examples

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

## End(Not run)
```

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

Description

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

Usage

```
drake_get_session_info(path = getwd(), search = TRUE,
  cache = drake::get_cache(path = path, search = search, verbose =
  verbose), verbose = 1L)
```

Arguments

path	Root directory of the drake project, or if search is TRUE, either the project root or a subdirectory of the project. Ignored if a cache is supplied.
search	Logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a cache is supplied.
cache	drake cache. See new_cache(). If supplied, path and search are ignored.
verbose	Logical or numeric, control printing to the console. • 0 or FALSE: print nothing. • 1 or TRUE: print only targets to build. • 2: plus checks and cache info. • 3: plus missing imports. • 4: plus all imports. • 5: plus execution and total build times for targets. • 6: plus notifications when targets are being stored.

Value

sessionInfo() of the last call to make()

See Also

diagnose(), cached(), readd(), drake_plan(), make()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Run the project, build the targets.
```

```
drake_get_session_info() # Get the cached sessionInfo() of the last make().
}
})

## End(Not run)
```

---

drake_ggraph *Show a* ggraph/ggplot2 *representation of your drake project.*

Description

This function requires packages ggplot2 and ggraph. Install them with `install.packages(c("ggplot2", "ggraph"))`.

Usage

```
drake_ggraph(config, build_times = "build", digits = 3,
  targets_only = FALSE, main = NULL, from = NULL, mode = c("out",
  "in", "all"), order = NULL, subset = NULL, make_imports = TRUE,
  from_scratch = FALSE, full_legend = FALSE, group = NULL,
  clusters = NULL, show_output_files = TRUE)
```

Arguments

config	A drake_config() configuration list. You can get one as a return value from make() as well.
build_times	Character string or logical. If character, the choices are 1. "build": runtime of the command plus the time it take to store the target or import. 2. "command": just the runtime of the command. 3. "none": no build times. If logical, build_times selects whether to show the times from 'build_times(..., type = "build")" or use no build times at all. See build_times() for details.
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.
main	Character string, title of the graph.
from	Optional collection of target/import names. If from is nonempty, the graph will restrict itself to a neighborhood of from. Control the neighborhood with mode and order.
mode	Which direction to branch out in the graph to create a neighborhood around from. Use "in" to go upstream, "out" to go downstream, and "all" to go both ways and disregard edge direction altogether.
order	How far to branch out to create a neighborhood around from. Defaults to as far as possible. If a target is in the neighborhood, then so are all of its custom file_out() files if show_output_files is TRUE. That means the actual graph order may be slightly greater than you might expect, but this ensures consistency between show_output_files = TRUE and show_output_files = FALSE.

subset	Optional character vector. Subset of targets/imports to display in the graph. Applied after from, mode, and order. Be advised: edges are only kept for adjacent nodes in subset. If you do not select all the intermediate nodes, edges will drop from the graph.
make_imports	Logical, whether to make the imports first. Set to FALSE to increase speed and risk using obsolete information.
from_scratch	Logical, whether to assume all the targets will be made from scratch on the next make(). Makes all targets outdated, but keeps information about build progress in previous make()s.
full_legend	Logical. If TRUE, all the node types are printed in the legend. If FALSE, only the node types used are printed in the legend.
group	Optional character scalar, name of the column used to group nodes into columns. All the columns names of your config$plan are choices. The other choices (such as "status") are column names in the nodes . To group nodes into clusters in the graph, you must also supply the clusters argument.
clusters	Optional character vector of values to cluster on. These values must be elements of the column of the nodes data frame that you specify in the group argument to drake_graph_info().
show_output_files	Logical, whether to include file_out() files in the graph.

Value

A ggplot2 object, which you can modify with more layers, show with plot(), or save as a file with ggsave().

See Also

vis_drake_graph(), sankey_drake_graph(), render_drake_ggraph()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
config <- drake_config(my_plan)
# Plot the network graph representation of the workflow.
if (requireNamespace("ggraph", quietly = TRUE)) {
  drake_ggraph(config) # Save to a file with `ggplot2::ggsave()`.
}
})

## End(Not run)
```

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

Description

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

Usage

```
drake_graph_info(config, from = NULL, mode = c("out", "in", "all"),
  order = NULL, subset = NULL, build_times = "build", digits = 3,
  targets_only = FALSE, font_size = 20, from_scratch = FALSE,
  make_imports = TRUE, full_legend = FALSE, group = NULL,
  clusters = NULL, show_output_files = TRUE, hover = FALSE)
```

Arguments

config	A drake_config() configuration list. You can get one as a return value from make() as well.
from	Optional collection of target/import names. If from is nonempty, the graph will restrict itself to a neighborhood of from. Control the neighborhood with mode and order.
mode	Which direction to branch out in the graph to create a neighborhood around from. Use "in" to go upstream, "out" to go downstream, and "all" to go both ways and disregard edge direction altogether.
order	How far to branch out to create a neighborhood around from. Defaults to as far as possible. If a target is in the neighborhood, then so are all of its custom file_out() files if show_output_files is TRUE. That means the actual graph order may be slightly greater than you might expect, but this ensures consistency between show_output_files = TRUE and show_output_files = FALSE.
subset	Optional character vector. Subset of targets/imports to display in the graph. Applied after from, mode, and order. Be advised: edges are only kept for adjacent nodes in subset. If you do not select all the intermediate nodes, edges will drop from the graph.
build_times	Character string or logical. If character, the choices are 1. "build": runtime of the command plus the time it take to store the target or import. 2. "command": just the runtime of the command. 3. "none": no build times. If logical, build_times selects whether to show the times from ‘build_times(..., type = "build")‘ or use no build times at all. See build_times() for details.
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.
font_size	Numeric, font size of the node labels in the graph

from_scratch — Logical, whether to assume all the targets will be made from scratch on the next make(). Makes all targets outdated, but keeps information about build progress in previous make()s.

make_imports — Logical, whether to make the imports first. Set to FALSE to increase speed and risk using obsolete information.

full_legend — Logical. If TRUE, all the node types are printed in the legend. If FALSE, only the node types used are printed in the legend.

group — Optional character scalar, name of the column used to group nodes into columns. All the columns names of your config$plan are choices. The other choices (such as "status") are column names in the nodes . To group nodes into clusters in the graph, you must also supply the clusters argument.

clusters — Optional character vector of values to cluster on. These values must be elements of the column of the nodes data frame that you specify in the group argument to drake_graph_info().

show_output_files — Logical, whether to include file_out() files in the graph.

hover — Logical, whether to show text (file contents, commands, etc.) when you hover your cursor over a node.

Value

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

See Also

vis_drake_graph()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (requireNamespace("visNetwork", quietly = TRUE)) {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
config <- drake_config(my_plan) # my_plan loaded with load_mtcars_example()
vis_drake_graph(config) # Jump straight to the interactive graph.
# Get a list of data frames representing the nodes, edges,
# and legend nodes of the visNetwork graph from vis_drake_graph().
raw_graph <- drake_graph_info(config = config)
# Choose a subset of the graph.
smaller_raw_graph <- drake_graph_info(
  config = config,
  from = c("small", "reg2"),
  mode = "in"
)
# Inspect the raw graph.
str(raw_graph)
# Use the data frames to plot your own custom visNetwork graph.
```

```
# For example, you can omit the legend nodes
# and change the direction of the graph.
library(visNetwork)
graph <- visNetwork(nodes = raw_graph$nodes, edges = raw_graph$edges)
visHierarchicalLayout(graph, direction = 'UD')
# Optionally visualize clusters.
config$plan$large_data <- grepl("large", config$plan$target)
graph <- drake_graph_info(
  config, group = "large_data", clusters = c(TRUE, FALSE))
tail(graph$nodes)
render_drake_graph(graph)
# You can even use clusters given to you for free in the `graph$nodes`
# data frame.
graph <- drake_graph_info(
  config, group = "status", clusters = "imported")
tail(graph$nodes)
render_drake_graph(graph)
}
}
})

## End(Not run)
```

`drake_hpc_template_file` *Write a template file for deploying work to a cluster / job scheduler.*

Description

See the example files from `drake_examples()` and `drake_example()` for example usage.

Usage

```
drake_hpc_template_file(file = drake::drake_hpc_template_files(),
  to = getwd(), overwrite = FALSE)
```

Arguments

`file`	Name of the template file, including the "tmpl" extension.
`to`	Character vector, where to write the file.
`overwrite`	Logical, whether to overwrite an existing file of the same name.

Value

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

See Also

`drake_hpc_template_files()`, `drake_examples()`, `drake_example()`, `shell_file()`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
# List the available template files.
drake_hpc_template_files()
# Write a SLURM template file from the SLURM example.
drake_hpc_template_file("slurm_batchtools.tmpl") # Writes slurm_batchtools.tmpl.
# library(future.batchtools) # nolint
# future::plan(batchtools_slurm, template = "slurm_batchtools.tmpl") # nolint
# make(my_plan, parallelism = "future", jobs = 2) # nolint
})

## End(Not run)
```

`drake_hpc_template_files` *List the available example template files for deploying work to a cluster / job scheduler.*

Description

See the example files from `drake_examples()` and `drake_example()` for example usage.

Usage

```
drake_hpc_template_files()
```

Value

A character vector of example template files that you can write with `drake_hpc_template_file()`.

See Also

`drake_hpc_template_file()`, `drake_examples()`, `drake_example()`, `shell_file()`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
# List the available template files.
drake_hpc_template_files()
# Write a SLURM template file from the SLURM example.
drake_hpc_template_file("slurm_batchtools.tmpl") # Writes slurm_batchtools.tmpl.
# library(future.batchtools) # nolint
# future::plan(batchtools_slurm, template = "slurm_batchtools.tmpl") # nolint
# make(my_plan, parallelism = "future", jobs = 2) # nolint
})
```

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

---

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

Description

A drake plan is a data frame with columns "target" and "command". Each target is an R object produced in your workflow, and each command is the R code to produce it. The "target" column has the names of the targets, and no duplicate elements are allowed. The "command" column is either a character vector of code strings or a list of language objects.

Usage

```
drake_plan(..., list = character(0), file_targets = NULL,
  strings_in_dots = NULL, tidy_evaluation = NULL, transform = TRUE,
  trace = FALSE, envir = parent.frame())
```

Arguments

...	A collection of symbols/targets with commands assigned to them. See the examples for details.
list	Deprecated
file_targets	Deprecated.
strings_in_dots	Deprecated.
tidy_evaluation	Logical, whether to use tidy evaluation such as quasiquotation when evaluating commands passed through the free-form ... argument.
transform	Logical, whether to transform the plan into a larger plan with more targets. This is still an experimental feature, so please check your workflow with vis_drake_graph() before running it with make(). Requires the transform and group fields identified by target(). See the examples for details.
trace	Logical, whether to add columns to show what happens during target transformations.
envir	Environment for tidy evaluation.

Details

drake has special syntax for generating large plans. Your code will look something like drake_plan(x = target(cmd, trar where f() is either map(), cross(), or combine() (similar to purrr::pmap(), tidy::crossing(), and dplyr::summarize(), respectively). These verbs mimic Tidyverse behavior to scale up existing plans to large numbers of targets. You can read about this interface at https://ropenscilabs.github.io/drake-manual/plans.html#create-large-plans-the-easy-way. # nolint

There is also special syntax for declaring input files, output files, and knitr reports so dependencies are properly accounted for (`file_in()`, `file_out()`, and `knitr_in()`, respectively.

Besides `"target"` and `"command"`, you may include optional columns in your workflow plan. For details, visit https://ropenscilabs.github.io/drake-manual/plans.html#special-custom-columns-in-your-pla

Value

A data frame of targets, commands, and optional custom columns.

Examples

```
test_with_dir("Contain side effects", {
# Create workflow plan data frames.
mtcars_plan <- drake_plan(
  write.csv(mtcars[, c("mpg", "cyl")], file_out("mtcars.csv")),
  value = read.csv(file_in("mtcars.csv"))
)
mtcars_plan
make(mtcars_plan) # Makes `mtcars.csv` and then `value`
head(readd(value))
# You can use knitr inputs too. See the top command below.
load_mtcars_example()
head(my_plan)
# The `knitr_in("report.Rmd")` tells `drake` to dive into the active
# code chunks to find dependencies.
# There, `drake` sees that `small`, `large`, and `coef_regression2_small`
# are loaded in with calls to `loadd()` and `readd()`.
deps_code("report.Rmd")

# Use transformations to generate large plans.
# This feature is experimental, so please
# check your workflow with `vis_drake_graph()`
# before running `make()`.
# Read more at
# <https://ropenscilabs.github.io/drake-manual/plans.html#create-large-plans-the-easy-way>. # nolint
drake_plan(
  data = target(
    simulate(nrows),
    transform = map(nrows = c(48, 64)),
    custom_column = 123
  ),
  reg = target(
    reg_fun(data),
   transform = cross(reg_fun = c(reg1, reg2), data)
  ),
  summ = target(
    sum_fun(data, reg),
   transform = cross(sum_fun = c(coef, residuals), reg)
  ),
  winners = target(
    min(summ),
    transform = combine(summ, .by = c(data, sum_fun))
```

```
  )
)

# Set trace = TRUE to show what happened during the transformation process.
drake_plan(
  data = target(
    simulate(nrows),
    transform = map(nrows = c(48, 64)),
    custom_column = 123
  ),
  reg = target(
    reg_fun(data),
   transform = cross(reg_fun = c(reg1, reg2), data)
  ),
  summ = target(
    sum_fun(data, reg),
   transform = cross(sum_fun = c(coef, residuals), reg)
  ),
  winners = target(
    min(summ),
    transform = combine(summ, .by = c(data, sum_fun))
  ),
  trace = TRUE
)

# You can create your own custom columns too.
# See ?triggers for more on triggers.
drake_plan(
  website_data = target(
    command = download_data("www.your_url.com"),
    trigger = "always",
    custom_column = 5
  ),
  analysis = analyze(website_data)
)

# Tidy evaluation can help generate super large plans.
sms <- rlang::syms(letters) # To sub in character args, skip this.
drake_plan(x = target(f(char), transform = map(char = !!sms)))
})
```

---

drake_plan_source *Show the code required to produce a given workflow plan data frame*

Description

You supply a plan, and drake_plan_source() supplies code to generate that plan. If you have the prettycode package, installed, you also get nice syntax highlighting in the console when you print it.

Usage

```
drake_plan_source(plan)
```

Arguments

plan	A workflow plan data frame (see drake_plan())

Value

a character vector of lines of text. This text is a call to drake_plan() that produces the plan you provide.

See Also

drake_plan()

Examples

```
plan <- drake::drake_plan(
  small_data = download_data("https://some_website.com"),
  large_data_raw = target(
    command = download_data("https://lots_of_data.com"),
    trigger = trigger(
      change = time_last_modified("https://lots_of_data.com"),
      command = FALSE,
      depend = FALSE
    ),
    timeout = 1e3
  )
)
print(plan)
if (requireNamespace("styler", quietly = TRUE)) {
  source <- drake_plan_source(plan)
  print(source) # Install the prettycode package for syntax highlighting.
  file <- tempfile() # Path to an R script to contain the drake_plan() call.
  writeLines(source, file) # Save the code to an R script.
}
```

---

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

Description

Wildcards are no longer recommended. Consider using transformations instead. Visit https://ropenscilabs.github.io/drake-manual/plans.html#large-plans for the details.

Usage

```
evaluate_plan(plan, rules = NULL, wildcard = NULL, values = NULL,
  expand = TRUE, rename = expand, trace = FALSE,
  columns = "command", sep = "_")
```

Arguments

plan	Workflow plan data frame, similar to one produced by drake_plan().
rules	Named list with wildcards as names and vectors of replacements as values. This is a way to evaluate multiple wildcards at once. When not NULL, rules overrules wildcard and values if not NULL.
wildcard	Character scalar denoting a wildcard placeholder.
values	Vector of values to replace the wildcard in the drake instructions. Will be treated as a character vector. Must be the same length as plan$command if expand is TRUE.
expand	If TRUE, create a new rows in the workflow plan data frame if multiple values are assigned to a single wildcard. If FALSE, each occurrence of the wildcard is replaced with the next entry in the values vector, and the values are recycled.
rename	Logical, whether to rename the targets based on the values supplied for the wildcards (based on values or rules).
trace	Logical, whether to add columns that trace the wildcard expansion process. These new columns indicate which targets were evaluated and with which wildcards.
columns	Character vector of names of columns to look for and evaluate the wildcards.
sep	Character scalar, separator for the names of the new targets generated. For example, in evaluate_plan(drake_plan(x = sqrt(y__)), list(y__ = 1:2), sep = "."), the names of the new targets are x.1 and x.2.

Details

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

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.

See Also

drake_plan()

Examples

```
# Create the part of the workflow plan for the datasets.
datasets <- drake_plan(
  small = simulate(5),
  large = simulate(50)
)
# Create a template workflow plan for the analyses.
methods <- drake_plan(
  regression1 = reg1(dataset__),
  regression2 = reg2(dataset__)
)
# Evaluate the wildcards in the template
# to produce the actual part of the workflow plan
# that encodes the analyses of the datasets.
# Create one analysis for each combination of dataset and method.
evaluate_plan(
  methods,
  wildcard = "dataset__",
  values = datasets$target
)
# Only choose some combinations of dataset and analysis method.
ans <- evaluate_plan(
  methods,
  wildcard = "dataset__",
  values = datasets$target,
  expand = FALSE
)
ans
# For the complete workflow plan, row bind the pieces together.
my_plan <- rbind(datasets, ans)
my_plan
# Wildcards for evaluate_plan() do not need the double-underscore suffix.
# Any valid symbol will do.
plan <- drake_plan(
  numbers = sample.int(n = `{Number}`, size = ..size)
)
evaluate_plan(
  plan,
  rules = list(
    "`{Number}`" = c(10, 13),
    ..size = c(3, 4)
  )
)
# Workflow plans can have multiple wildcards.
# Each combination of wildcard values will be used
# Except when expand is FALSE.
x <- drake_plan(draws = sample.int(n = N, size = Size))
evaluate_plan(x, rules = list(N = 10:13, Size = 1:2))
# You can use wildcards on columns other than "command"
evaluate_plan(
  drake_plan(
    x = target("1 + 1", cpu = "any"),
```

```
    y = target("sqrt(4)", cpu = "always"),
    z = target("sqrt(16)", cpu = "any")
  ),
  rules = list(always = 1:2),
  columns = c("command", "cpu")
)
# With the `trace` argument,
# you can generate columns that show how the wildcards
# were evaluated.
plan <- drake_plan(x = sample.int(n__), y = sqrt(n__))
plan <- evaluate_plan(plan, wildcard = "n__", values = 1:2, trace = TRUE)
print(plan)
# With the `trace` argument,
# you can generate columns that show how the wildcards
# were evaluated. Then you can visualize the wildcard groups
# as clusters.
plan <- drake_plan(x = sqrt(n__), y = sample.int(n__))
plan <- evaluate_plan(plan, wildcard = "n__", values = 1:2, trace = TRUE)
print(plan)
cache <- storr::storr_environment()
config <- drake_config(plan, cache = cache)
## Not run:
if (requireNamespace("visNetwork", quietly = TRUE)) {
vis_drake_graph(config, group = "n__", clusters = "1")
vis_drake_graph(config, group = "n__", clusters = c("1", "2"))
make(plan, targets = c("x_1", "y_2"), cache = cache)
# Optionally cluster on columns supplied by `drake_graph_info()$nodes`.
vis_drake_graph(config, group = "status", clusters = "up to date")
}

## End(Not run)
```

`expand_plan` *Create replicates of targets.*

Description

`expand_plan()` is no longer recommended. Consider using transformations instead. Visit https://ropenscilabs.github.io/drake-manual/plans.html#large-plans for the details.

Usage

```
expand_plan(plan, values = NULL, rename = TRUE, sep = "_",
  sanitize = TRUE)
```

Arguments

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

rename	Logical, whether to rename the targets based on the values. See the examples for a demo.
sep	Character scalar, delimiter between the original target names and the values to append to create the new target names. Only relevant when rename is TRUE.
sanitize	Logical, whether to sanitize the plan.

Details

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

Value

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

See Also

drake_plan()

Examples

```
# Create the part of the workflow plan for the datasets.
datasets <- drake_plan(
  small = simulate(5),
  large = simulate(50)
)
# Create replicates. If you want repeat targets,
# this is convenient.
expand_plan(datasets, values = c("rep1", "rep2", "rep3"))
# Choose whether to rename the targets based on the values.
expand_plan(datasets, values = 1:3, rename = TRUE)
expand_plan(datasets, values = 1:3, rename = FALSE)
```

---

expose_imports *Expose all the imports in a package so* make() *can detect all the package's nested functions.*

Description

When drake analyzes the functions in your environment, it understands that some of your functions are nested inside other functions. It dives into nested function after nested function in your environment so that if an inner function changes, targets produced by the outer functions will become out of date. However, drake stops searching as soon as it sees a function from a package. This keeps projects from being too brittle, but it is sometimes problematic. You may want to strongly depend on a package's internals. In fact, you may want to wrap your data analysis project itself in a formal R package, so you want all your functions to be reproducibly tracked.

To make all a package's functions available to be tracked as dependencies, use the expose_imports() function. See the examples in this help file for a demonstration.

Usage

```
expose_imports(package, character_only = FALSE, envir = parent.frame(),
  jobs = 1)
```

Arguments

package	Name of the package, either a symbol or a string, depending on character_only.
character_only	Logical, whether to interpret package as a character string or a symbol (quoted vs unquoted).
envir	Environment to load the exposed package imports. You will later pass this envir to make().
jobs	Number of parallel jobs for the parallel processing of the imports.

Details

Thanks to Jasper Clarkberg for the idea that makes this function work.

Value

The environment that the exposed imports are loaded into. Defaults to your R workspace.

Examples

```
## Not run:
test_with_dir("contain this example's side effects", {
# Suppose you have a workflow that uses the `digest()` function,
# which computes the hash of an object.

library(digest) # Has the digest() function.
g <- function(x) {
  digest(x)
}
f <- function(x) {
  g(x)
}
plan <- drake_plan(x = f(1))

# Here are the reproducibly tracked objects in the workflow.
config <- drake_config(plan)
tracked(config)

# But the digest() function has dependencies too.
head(deps_code(digest))

# Why doesn't `drake` import them? Because it knows `digest()`
# is from a package, and it doesn't usually dive into functions
# from packages. We need to call expose_imports() to expose
# a package's inner functions.

expose_imports(digest)
```

```
config <- drake_config(plan)
new_objects <- tracked(config)
head(new_objects, 10)
length(new_objects)

# Now when you call `make()`, `drake` will dive into `digest`
# to import dependencies.

cache <- storr::storr_environment() # just for examples
make(plan, cache = cache)
head(cached(cache = cache), 10)
length(cached(cache = cache))

# Why would you want to expose a whole package like this?
# Because you may want to wrap up your data science project
# as a formal R package. In that case, `expose_imports()`
# tells `drake` to reproducibly track all of your code,
# not just the exported API functions you mention in
# workflow plan commands.

# Note: if you use `digest::digest()`` instead of just `digest()`,
# `drake` does not dive into the function body anymore.
g <- function(x) {
  digest::digest(x) # Was previously just digest()
}
config <- drake_config(plan)
tracked(config)
})

## End(Not run)
```

`failed` *List failed targets. to* `make()`.

Description

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

Usage

```
failed(path = getwd(), search = TRUE, cache = drake::get_cache(path =
  path, search = search, verbose = verbose), verbose = 1L,
  upstream_only = NULL)
```

Arguments

`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project. Ignored if a `cache` is supplied.

search	Logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a cache is supplied.
cache	drake cache. See new_cache(). If supplied, path and search are ignored.
verbose	Logical or numeric, control printing to the console. • 0 or FALSE: print nothing. • 1 or TRUE: print only targets to build. • 2: plus checks and cache info. • 3: plus missing imports. • 4: plus all imports. • 5: plus execution and total build times for targets. • 6: plus notifications when targets are being stored.
upstream_only	Deprecated.

Value

A character vector of target names.

See Also

running(), make()

Examples

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

## End(Not run)
```

---

file_in *Declare the file inputs of a workflow plan command.*

Description

Use this function to help write the commands in your workflow plan data frame. See the examples for a full explanation.

Usage

```
file_in(...)
```

Arguments

...	Character strings. File paths of input files to a command in your workflow plan data frame.

Value

A character vector of declared input file paths.

See Also

file_out(), knitr_in(), ignore()

Examples

```
## Not run:
test_with_dir("Contain side effects", {
# The `file_out()` and `file_in()` functions
# just takes in strings and returns them.
file_out("summaries.txt")
# Their main purpose is to orchestrate your custom files
# in your workflow plan data frame.
suppressWarnings(
  plan <- drake_plan(
    write.csv(mtcars, file_out("mtcars.csv")),
    contents = read.csv(file_in("mtcars.csv"))
  )
)
plan
# drake knows "\"mtcars.csv\"" is the first target
# and a dependency of `contents`. See for yourself:
make(plan)
file.exists("mtcars.csv")
# See also `knitr_in()`. `knitr_in()` is like `file_in()`
# except that it analyzes active code chunks in your `knitr`
# source file and detects non-file dependencies.
# That way, updates to the right dependencies trigger rebuilds
# in your report.
})

## End(Not run)
```

file_out *Declare the file outputs of a workflow plan command.*

Description

Use this function to help write the commands in your workflow plan data frame. You can only specify one file output per command. See the examples for a full explanation.

Usage

```
file_out(...)
```

Arguments

...	Character vector of output file paths.

Value

A character vector of declared output file paths.

See Also

file_in(), knitr_in(), ignore()

Examples

```
## Not run:
test_with_dir("Contain side effects", {
# The `file_out()` and `file_in()` functions
# just takes in strings and returns them.
file_out("summaries.txt", "output.csv")
# Their main purpose is to orchestrate your custom files
# in your workflow plan data frame.
suppressWarnings(
  plan <- drake_plan(
    write.csv(mtcars, file_out("mtcars.csv")),
    contents = read.csv(file_in("mtcars.csv"))
  )
)
plan
# drake knows "\"mtcars.csv\"" is the first target
# and a dependency of `contents`. See for yourself:
make(plan)
file.exists("mtcars.csv")
# See also `knitr_in()`. `knitr_in()` is like `file_in()`
# except that it analyzes active code chunks in your `knitr`
# source file and detects non-file dependencies.
# That way, updates to the right dependencies trigger rebuilds
# in your report.
```

```
})

## End(Not run)
```

`file_store` *Tell* drake *that you want information on a* file *(target or import), not an ordinary object.*

Description

This function simply wraps literal double quotes around the argument `x` so `drake` knows it is the name of a file. Use when you are calling functions like `deps_code()`: for example, `deps_code(file_store("report.md"))`. See the examples for details. Internally, `drake` wraps the names of file targets/imports inside literal double quotes to avoid confusion between files and generic R objects.

Usage

```
file_store(x)
```

Arguments

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

Value

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

Examples

```
# Wraps the string in single quotes.
file_store("my_file.rds") # "'my_file.rds'"
## Not run:
test_with_dir("contain side effects", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Run the workflow to build the targets
list.files() # Should include input "report.Rmd" and output "report.md".
head(readd(small)) # You can use symbols for ordinary objects.
# But if you want to read cached info on files, use `file_store()`.
readd(file_store("report.md"), character_only = TRUE) # File fingerprint.
deps_code(file_store("report.Rmd"))
config <- drake_config(my_plan)
deps_profile(
  file_store("report.Rmd"),
  config = config,
  character_only = TRUE
)
}
```

```
  })

## End(Not run)
```

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

Description

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

Usage

```
find_cache(path = getwd(), dir = NULL, directory = NULL)
```

Arguments

`path`	Starting path for search back for the cache. Should be a subdirectory of the drake project.
`dir`	Character, name of the folder containing the cache.
`directory`	Deprecated. Use `dir`.

Value

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

See Also

`drake_plan()`, `make()`,

Examples

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

## End(Not run)
```

from_plan *Get a target's plan info from inside a plan's command.*

Description

Call this function inside the commands in your plan to get an entry from any column in the plan. Changes to custom columns referred to this way (for example, a in `drake_plan(x = target(from_plan("a"), a = 123))`) do not invalidate targets, so be careful. Only use `from_plan()` to reference data that does not actually affect the output value of the target. For example, you might use `from_plan()` to set the number of parallel workers within a target: `drake_plan(x = target(mclapply(..., from_plan("cores"), cores = 4))`. Now, if you change the `cores` column of the plan, the parallelism will change, but the target `x` will stay up to date.

Usage

```
from_plan(column)
```

Arguments

`column`	Character, name of a column in your `drake` plan.

Value

`plan[target, column]`, where `plan` is your workflow plan data frame, `target` is the target being built, and `column` is the name of the column of the plan you provide.

See Also

`drake_envir()`

Examples

```
plan <- drake_plan(my_target = target(from_plan("a"), a = "a_value"))
plan

cache <- storr::storr_environment()
make(plan, cache = cache, session_info = FALSE)
readd(my_target, cache = cache)

# Why do we care?
# Because this is a good way to apply parallel computing within targets
# and keep them up to date even when we change
# the number of "cores"

plan <- drake_plan(
  a = target(
    parallel::mclapply(1:8, sqrt, mc.cores = from_plan("cores")),
    cores = 4
  ),
```

```
  b = target(
    parallel::mclapply(1:4, sqrt, mc.cores = from_plan("cores")),
    cores = 2
  )
)

plan

# If make(plan, parallelism = "loop") fails,
# try make(plan, lock_envir = FALSE)
# or another parallel computing option like parLapply()
# or furrr::future_map().
# See https://github.com/ropensci/drake/issues/675#issuecomment-454403818
# and the ensuing comments for a discussion.

# But usually, parallelism *among* targets happens through a cluster.
# drake_hpc_template_file("slurm_clustermq.tmpl") # Edit by hand. # nolint
# options(
#   clustermq.scheduler = "slurm",
#   clustermq.template = "slurm_clustermq.tmpl",
# )
# make(plan, parallelism = "clustermq", jobs = 2) # nolint

# Now, if you change the `cores` column of the plan, the parallelism will
# change, but the targets will stay up to date.
plan$cores <- c(1, 1)
plan

# make(plan) # nolint
```

`gather_by` *Gather multiple groupings of targets*

Description

`gather_by()` is no longer recommended. Consider using transformations instead. Visit https://ropenscilabs.github.io/drake-manual/plans.html#large-plans for the details.

Usage

```
gather_by(plan, ..., prefix = "target", gather = "list",
  append = TRUE, filter = NULL, sep = "_")
```

Arguments

`plan`	Workflow plan data frame of prespecified targets.
`...`	Symbols, columns of `plan` to define target groupings. A `gather_plan()` call is applied for each grouping. Groupings with all `NA`s in the selector variables are ignored.

prefix	Character, prefix for naming the new targets. Suffixes are generated from the values of the columns specified in `...`.
gather	Function used to gather the targets. Should be one of `list(...)`, `c(...)`, `rbind(...)`, or similar.
append	Logical. If `TRUE`, the output will include the original rows in the `plan` argument. If `FALSE`, the output will only include the new targets and commands.
filter	An expression like you would pass to `dplyr::filter()`. The rows for which `filter` evaluates to `TRUE` will be gathered, and the rest will be excluded from gathering. Why not just call `dplyr::filter()` before `gather_by()`? Because `gather_by(append = TRUE, filter = my_column == "my_value")` gathers on some targets while including all the original targets in the output. See the examples for a demonstration.
sep	Character scalar, delimiter for creating the names of new targets.

Details

Perform several calls to `gather_plan()` based on groupings from columns in the plan, and then row-bind the new targets to the plan.

Value

A workflow plan data frame.

See Also

`drake_plan()`

Examples

```
plan <- drake_plan(
  data = get_data(),
  informal_look = inspect_data(data, mu = mu__),
  bayes_model = bayesian_model_fit(data, prior_mu = mu__)
)
plan <- evaluate_plan(plan, rules = list(mu__ = 1:2), trace = TRUE)
plan
gather_by(plan, mu___from, gather = "rbind")
gather_by(plan, mu___from, append = TRUE)
gather_by(plan, mu___from, append = FALSE)
gather_by(plan, mu__, mu___from, prefix = "x")
gather_by(plan) # Gather everything and append a row.
# You can filter out the informal_look_* targets beforehand
# if you only want the bayes_model_* ones to be gathered.
# The advantage here is that if you also need `append = TRUE`,
# only the bayes_model_* targets will be gathered, but
# the informal_look_* targets will still be included
# in the output.
gather_by(
  plan,
  mu___from,
```

```
  append = TRUE,
  filter = mu___from == "bayes_model"
)
```

---

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

Description

gather_plan() is no longer recommended. Consider using transformations instead. Visit https://ropenscilabs.github.io/drake-manual/plans.html#large-plans for the details.

Usage

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

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.
append	Logical. If TRUE, the output will include the original rows in the plan argument. If FALSE, the output will only include the new targets and commands.

Details

Creates a new workflow plan to aggregate existing targets in the supplied plan.

Value

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

See Also

drake_plan()

Examples

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

---

get_cache *Get the default cache of a* drake *project.*

Description

Only works if the cache is in a folder called `.drake/`. See the description of the `path` argument for details.

Usage

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

Arguments

`path`	Character, either the root file path of a `drake` project or a folder containing the root (top-level working directory where you plan to call `make()`). If this is too confusing, feel free to just use `storr::storr_rds()` to get the cache. If `search = FALSE`, `path` must be the root. If `search = TRUE`, you can specify any subdirectory of the project. Let's say `"/home/you/my_project"` is the root. The following are equivalent and correct:

- `get_cache(path = "/home/you/my_project", search = FALSE)`
- `get_cache(path = "/home/you/my_project", search = TRUE)`
- `get_cache(path = "/home/you/my_project/subdir/x", search = TRUE)`
- `get_cache(path = "/home/you/my_project/.drake", search = TRUE)`
- `get_cache(path = "/home/you/my_project/.drake/keys", search = TRUE)`

`search` Logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a `cache` is supplied.

`verbose` Logical or numeric, control printing to the console.

- `0` or `FALSE`: print nothing.
- `1` or `TRUE`: print only targets to build.
- `2`: plus checks and cache info.
- `3`: plus missing imports.
- `4`: plus all imports.
- `5`: plus execution and total build times for targets.
- `6`: plus notifications when targets are being stored.

`force` Deprecated.

`fetch_cache` Character vector containing lines of code. The purpose of this code is to fetch the `storr` cache with a command like `storr_rds()` or `storr_dbi()`, but customized. This feature is experimental.

`console_log_file`
Character scalar, connection object (such as `stdout()`) or `NULL`. If `NULL`, console output will be printed to the R console using `message()`. If a character scalar, `console_log_file` should be the name of a flat file, and console output will be appended to that file. If a connection object (e.g. `stdout()`) warnings and messages will be sent to the connection. For example, if `console_log_file` is `stdout()`, warnings and messages are printed to the console in real time (in addition to the usual in-bulk printing after each target finishes).

Value

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

See Also

`new_cache()`, `drake_config()`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
clean(destroy = TRUE)
# No cache is available.
get_cache() # NULL
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Run the project, build the targets.
x <- get_cache() # Now, there is a cache.
```

```
y <- storr::storr_rds(".drake") # Equivalent.
# List the objects readable from the cache with readd().
x$list()
}
})

## End(Not run)
```

`ignore` *Ignore components of commands and imported functions.*

Description

In a command in the workflow plan or the body of an imported function, you can `ignore(some_code)` to

1. Force `drake` to not track dependencies in `some_code`, and
2. Ignore any changes in `some_code` when it comes to deciding which target are out of date.

Usage

```
ignore(x = NULL)
```

Arguments

`x`	Code to ignore.

Value

The argument.

See Also

`file_in()`, `file_out()`, `knitr_in()`

Examples

```
## Not run:
test_with_dir("Contain side effects", {
# Normally, `drake` reacts to changes in dependencies.
x <- 4
make(plan = drake_plan(y = sqrt(x)))
x <- 5
make(plan = drake_plan(y = sqrt(x)))
make(plan = drake_plan(y = sqrt(4) + x))
# But not with ignore().
make(plan = drake_plan(y = sqrt(4) + ignore(x))) # Builds y.
x <- 6
make(plan = drake_plan(y = sqrt(4) + ignore(x))) # Skips y.
```

```
make(plan = drake_plan(y = sqrt(4) + ignore(x + 1))) # Skips y.
# What about imported functions?
f <- function(x) sqrt(4) + ignore(x + 1)
make(plan = drake_plan(x = f(2)))
readd(x)
f <- function(x) sqrt(4) + ignore(x + 2)
make(plan = drake_plan(x = f(2)))
readd(x)
f <- function(x) sqrt(5) + ignore(x + 2)
make(plan = drake_plan(x = f(2)))
readd(x)
})

## End(Not run)
```

`knitr_in` *Declare the* `knitr/rmarkdown` *source files of a workflow plan command.*

Description

Use this function to help write the commands in your workflow plan data frame. See the examples for a full explanation.

Usage

```
knitr_in(...)
```

Arguments

`...`	Character strings. File paths of `knitr/rmarkdown` source files supplied to a command in your workflow plan data frame.

Value

A character vector of declared input file paths.

See Also

`file_in()`, `file_out()`, `ignore()`

Examples

```
## Not run:
test_with_dir("Contain side effects", {
if (suppressWarnings(require("knitr"))) {
# `knitr_in()` is like `file_in()`
# except that it analyzes active code chunks in your `knitr`
# source file and detects non-file dependencies.
# That way, updates to the right dependencies trigger rebuilds
```

```
# in your report.
# The mtcars example (`drake_example("mtcars")`)
# already has a demonstration
load_mtcars_example()
make(my_plan)
# Now how did drake magically know that
# `small`, `large`, and `coef_regression2_small` were
# dependencies of the output file `report.md`?
# because the command in the workflow plan had
# `knitr_in("report.Rmd")` in it, so drake knew
# to analyze the active code chunks. There, it spotted
# where `small`, `large`, and `coef_regression2_small`
# were read from the cache using calls to `loadd()` and `readd()`.
}
})

## End(Not run)
```

`legend_nodes` *Create the nodes data frame used in the legend of the graph visualizations.*

Description

Output a `visNetwork`-friendly data frame of nodes. It tells you what the colors and shapes mean in the graph visualizations.

Usage

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

Arguments

`font_size`	Font size of the node label text.

Value

A data frame of legend nodes for the graph visualizations.

Examples

```
## Not run:
# Show the legend nodes used in graph visualizations.
# For example, you may want to inspect the color palette more closely.
if (requireNamespace("visNetwork", quietly = TRUE)) {
visNetwork::visNetwork(nodes = legend_nodes()) # nolint
}

## End(Not run)
```

`load_mtcars_example` *Load the mtcars example.*

Description

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

Usage

```
load_mtcars_example(envir = parent.frame(), report_file = NULL,
  overwrite = FALSE, force = FALSE)
```

Arguments

`envir`	The environment to load the example into. Defaults to your workspace. For an insulated workspace, set `envir = new.env(parent = globalenv())`.
`report_file`	Where to write the report file. Deprecated. In a future release, the report file will always be `report.Rmd` and will always be written to your working directory (current default).
`overwrite`	Logical, whether to overwrite an existing file `report.Rmd`.
`force`	Deprecated.

Details

Use `drake_example("mtcars")` to get the code for the mtcars example. The included R script is a detailed, heavily-commented walkthrough. The chapter of the user manual at https://ropenscilabs.github.io/drake-manual/mtcars.html # nolint also walks through the mtcars example. This function also writes/overwrites the file, `report.Rmd`.

Value

Nothing.

See Also

`clean_mtcars_example()` `drake_examples()`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
# Populate your workspace and write 'report.Rmd'.
load_mtcars_example() # Get the code: drake_example("mtcars")
```

```
# Check the dependencies of an imported function.
deps_code(reg1)
# Check the dependencies of commands in the workflow plan.
deps_code(my_plan$command[1])
deps_code(my_plan$command[4])
# Plot the interactive network visualization of the workflow.
config <- drake_config(my_plan)
outdated(config) # Which targets are out of date?
# Run the workflow to build all the targets in the plan.
make(my_plan)
outdated(config) # Everything should be up to date.
# For the reg2() model on the small dataset,
# the p-value is so small that there may be an association
# between weight and fuel efficiency after all.
readd(coef_regression2_small)
# Clean up the example.
clean_mtcars_example()
}
})

## End(Not run)
```

---

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

Description

This is the central, most important function of the drake package. It runs all the steps of your workflow in the correct order, skipping any work that is already up to date. See https://github.com/ropensci/drake/blob/master/README.md#documentation for an overview of the documentation.

Usage

```
make(plan, targets = NULL, envir = parent.frame(), verbose = 1L,
  hook = NULL, cache = drake::get_cache(verbose = verbose,
  console_log_file = console_log_file), fetch_cache = NULL,
  parallelism = "loop", jobs = 1L, jobs_preprocess = 1L,
  packages = rev(.packages()), lib_loc = NULL,
  prework = character(0), prepend = NULL, command = NULL,
  args = NULL, recipe_command = NULL, log_progress = TRUE,
  skip_targets = FALSE, timeout = NULL, cpu = Inf, elapsed = Inf,
  retries = 0, force = FALSE, graph = NULL,
  trigger = drake::trigger(), skip_imports = FALSE,
  skip_safety_checks = FALSE, config = NULL, lazy_load = "eager",
  session_info = TRUE, cache_log_file = NULL, seed = NULL,
  caching = "master", keep_going = FALSE, session = NULL,
  pruning_strategy = NULL, makefile_path = NULL,
  console_log_file = NULL, ensure_workers = TRUE,
```

```
garbage_collection = FALSE, template = list(), sleep = function(i)
0.01, hasty_build = NULL, memory_strategy = c("speed", "memory",
"lookahead"), layout = NULL, lock_envir = TRUE)
```

Arguments

plan	Workflow plan data frame. A workflow plan data frame is a data frame with a `target` column and a `command` column. (See the details in the `drake_plan()` help file for descriptions of the optional columns.) Targets are the objects that drake generates, and commands are the pieces of R code that produce them. You can create and track custom files along the way (see `file_in()`, `file_out()`, and `knitr_in()`). Use the function `drake_plan()` to generate workflow plan data frames.
targets	Character vector, names of targets to build. Dependencies are built too. Together, the `plan` and `targets` comprise the workflow network (i.e. the `graph` argument). Changing either will change the network.
envir	Environment to use. Defaults to the current workspace, so you should not need to worry about this most of the time. A deep copy of `envir` is made, so you don't need to worry about your workspace being modified by `make`. The deep copy inherits from the global environment. Wherever necessary, objects and functions are imported from `envir` and the global environment and then reproducibly tracked as dependencies.
verbose	Logical or numeric, control printing to the console. • `0` or `FALSE`: print nothing. • `1` or `TRUE`: print only targets to build. • `2`: plus checks and cache info. • `3`: plus missing imports. • `4`: plus all imports. • `5`: plus execution and total build times for targets. • `6`: plus notifications when targets are being stored.
hook	Deprecated.
cache	drake cache as created by `new_cache()`. See also `get_cache()`.
fetch_cache	Deprecated.
parallelism	Character scalar, type of parallelism to use. For detailed explanations, see the high-performance computing chapter of the user manual. You could also supply your own scheduler function if you want to experiment or aggressively optimize. The function should take a single `config` argument (produced by `drake_config()`). Existing examples from `drake`'s internals are the `backend_*()` functions: • `backend_loop()` • `backend_clustermq()` • `backend_future()` • `backend_hasty()` (unofficial) However, this functionality is really a back door and should not be used for production purposes unless you really know what you are doing and you are willing to suffer setbacks whenever `drake`'s unexported core functions are updated.

jobs — Maximum number of parallel workers for processing the targets. You can experiment with predict_runtime() to help decide on an appropriate number of jobs. For details, visit https://ropenscilabs.github.io/drake-manual/time.html.

jobs_preprocess — Number of parallel jobs for processing the imports and doing other preprocessing tasks.

packages — Character vector packages to load, in the order they should be loaded. Defaults to rev(.packages()), so you should not usually need to set this manually. Just call library() to load your packages before make(). However, sometimes packages need to be strictly forced to load in a certain order, especially if parallelism is "Makefile". To do this, do not use library() or require() or loadNamespace() or attachNamespace() to load any libraries beforehand. Just list your packages in the packages argument in the order you want them to be loaded.

lib_loc — Character vector, optional. Same as in library() or require(). Applies to the packages argument (see above).

prework — Expression (language object), list of expressions, or character vector. Code to run right before targets build. Called only once if parallelism is "loop" and once per target otherwise. This code can be used to set global options, etc.

prepend — Deprecated.

command — Deprecated.

args — Deprecated.

recipe_command — Deprecated.

log_progress — Logical, whether to log the progress of individual targets as they are being built. Progress logging creates a lot of little files in the cache, and it may make builds a tiny bit slower. So you may see gains in storage efficiency and speed with make(..., log_progress = FALSE).

skip_targets — Logical, whether to skip building the targets in plan and just import objects and files.

timeout — deprecated. Use elapsed and cpu instead.

cpu — Same as the cpu argument of setTimeLimit(). Seconds of cpu time before a target times out. Assign target-level cpu timeout times with an optional cpu column in plan.

elapsed — Same as the elapsed argument of setTimeLimit(). Seconds of elapsed time before a target times out. Assign target-level elapsed timeout times with an optional elapsed column in plan.

retries — Number of retries to execute if the target fails. Assign target-level retries with an optional retries column in plan.

force — Logical. If FALSE (default) then drake imposes checks if the cache was created with an old and incompatible version of drake. If there is an incompatibility, make() stops to give you an opportunity to downgrade drake to a compatible version rather than rerun all your targets from scratch.

graph — An igraph object from the previous make(). Supplying a pre-built graph could save time.

`trigger` Name of the trigger to apply to all targets. Ignored if `plan` has a `trigger` column. See `trigger()` for details.

`skip_imports` Logical, whether to totally neglect to process the imports and jump straight to the targets. This can be useful if your imports are massive and you just want to test your project, but it is bad practice for reproducible data analysis. This argument is overridden if you supply your own `graph` argument.

`skip_safety_checks` Logical, whether to skip the safety checks on your workflow. Use at your own peril.

`config` A list generated by `drake_config()`. `drake_config()` collects and sanitizes the multitude of parameters and settings that `make()` needs to do its job: the plan, packages, the environment of functions and initial data objects, parallel computing instructions, verbosity level, etc. Other functions such as `outdated()`, `vis_drake_graph()`, and `predict_runtime()` require output from `drake_config()` for the `config` argument. If you supply a `drake_config()` object to the `config` argument of `make()`, then `drake` will ignore all the other arguments because it already has everything it needs in `config`.

`lazy_load` Either a character vector or a logical. Choices:

- `"eager"`: no lazy loading. The target is loaded right away with `assign()`.
- `"promise"`: lazy loading with `delayedAssign()`
- `"bind"`: lazy loading with active bindings: `bindr::populate_env()`.
- `TRUE`: same as `"promise"`.
- `FALSE`: same as `"eager"`.

`lazy_load` should not be `"promise"` for `"parLapply"` parallelism combined with `jobs` greater than 1. For local multi-session parallelism and lazy loading, try `library(future); future::plan(multisession)` and then `make(..., parallelism = "future_`

If `lazy_load` is `"eager"`, drake prunes the execution environment before each target/stage, removing all superfluous targets and then loading any dependencies it will need for building. In other words, drake prepares the environment in advance and tries to be memory efficient. If `lazy_load` is `"bind"` or `"promise"`, drake assigns promises to load any dependencies at the last minute. Lazy loading may be more memory efficient in some use cases, but it may duplicate the loading of dependencies, costing time.

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

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

seed
: Integer, the root pseudo-random number generator seed to use for your project. In `make()`, drake generates a unique local seed for each target using the global seed and the target name. That way, different pseudo-random numbers are generated for different targets, and this pseudo-randomness is reproducible.

  To ensure reproducibility across different R sessions, `set.seed()` and `.Random.seed` are ignored and have no affect on drake workflows. Conversely, `make()` does not usually change `.Random.seed`, even when pseudo-random numbers are generated. The exceptions to this last point are `make(parallelism = "clustermq")` and `make(parallelism = "clustermq_staged")`, because the clustermq package needs to generate random numbers to set up ports and sockets for ZeroMQ.

  On the first call to `make()` or `drake_config()`, drake uses the random number generator seed from the `seed` argument. Here, if the `seed` is `NULL` (default), drake uses a `seed` of `0`. On subsequent `make()`s for existing projects, the project's cached seed will be used in order to ensure reproducibility. Thus, the `seed` argument must either be `NULL` or the same seed from the project's cache (usually the `.drake/` folder). To reset the random number generator seed for a project, use `clean(destroy = TRUE)`.

caching
: Character string, only applies to `"clustermq"`, `"clustermq_staged"`, and `"future"` parallel backends. The `caching` argument can be either `"master"` or `"worker"`.

  - `"master"`: Targets are built by remote workers and sent back to the master process. Then, the master process saves them to the cache (`config$cache`, usually a file system `storr`). Appropriate if remote workers do not have access to the file system of the calling R session. Targets are cached one at a time, which may be slow in some situations.
  - `"worker"`: Remote workers not only build the targets, but also save them to the cache. Here, caching happens in parallel. However, remote workers need to have access to the file system of the calling R session. Transferring target data across a network can be slow.

keep_going
: Logical, whether to still keep running `make()` if targets fail.

session
: Deprecated. Has no effect now.

pruning_strategy
: Deprecated. See `memory_strategy`.

makefile_path
: Path to the `Makefile` for `make(parallelism = "Makefile")`. If you set this argument to a non-default value, you are responsible for supplying this same path to the `args` argument so `make` knows where to find it. Example: `make(parallelism = "Makefile", makefile_path = ".drake/.makefile", command = "make", a`

console_log_file
: Character scalar, connection object (such as `stdout()`) or `NULL`. If `NULL`, console output will be printed to the R console using `message()`. If a character scalar, `console_log_file` should be the name of a flat file, and console output will be appended to that file. If a connection object (e.g. `stdout()`) warnings and messages will be sent to the connection. For example, if `console_log_file` is `stdout()`, warnings and messages are printed to the console in real time (in addition to the usual in-bulk printing after each target finishes).

ensure_workers Logical, whether the master process should wait for the workers to post before assigning them targets. Should usually be TRUE. Set to FALSE for make(parallelism = "future_lapply (n > 1) when combined with future::plan(future::sequential). This argument only applies to parallel computing with persistent workers (make(parallelism = x), where x could be "mclapply", "parLapply", or "future_lapply").

garbage_collection
Logical, whether to call gc() each time a target is built during make().

template A named list of values to fill in the {{ ... }} placeholders in template files (e.g. from drake_hpc_template_file()). Same as the template argument of clustermq::Q() and clustermq::workers. Enabled for clustermq only (make(parallelism = "clustermq_staged")), not future or batchtools so far. For more information, see the clustermq package: https://github.com/mschubert/clustermq. Some template placeholders such as {{ job_name }} and {{ n_jobs }} cannot be set this way.

sleep In its parallel processing, drake uses a central master process to check what the parallel workers are doing, and for the affected high-performance computing workflows, wait for data to arrive over a network. In between loop iterations, the master process sleeps to avoid throttling. The sleep argument to make() and drake_config() allows you to customize how much time the master process spends sleeping.

The sleep argument is a function that takes an argument i and returns a numeric scalar, the number of seconds to supply to Sys.sleep() after iteration i of checking. (Here, i starts at 1.) If the checking loop does something other than sleeping on iteration i, then i is reset back to 1.

To sleep for the same amount of time between checks, you might supply something like function(i) 0.01. But to avoid consuming too many resources during heavier and longer workflows, you might use an exponential back-off: say, function(i) { 0.1 + 120 * pexp(i - 1, rate = 0.01) }.

hasty_build A user-defined function. In "hasty mode" (make(parallelism = "hasty")) this is the function that evaluates a target's command and returns the resulting value. The hasty_build argument has no effect if parallelism is any value other than "hasty".

The function you pass to hasty_build must have arguments target and config. Here, target is a character scalar naming the target being built, and config is a configuration list of runtime parameters generated by drake_config().

memory_strategy
Character scalar, name of the strategy drake uses to manage targets in memory. For more direct control over which targets drake keeps in memory, see the help file examples of drake_envir(). The memory_strategy argument to make() and drake_config() is an attempt at an automatic catch-all solution. These are the choices.

- "speed": Once a target is loaded in memory, just keep it there. This choice maximizes speed and hogs memory.
- "memory": Just before building each new target, unload everything from memory except the target's direct dependencies. This option conserves memory, but it sacrifices speed because each new target needs to reload any previously unloaded targets from storage.

- "lookahead": Just before building each new target, search the dependency graph to find targets that will not be needed for the rest of the current make() session. In this mode, targets are only in memory if they need to be loaded, and we avoid superfluous reads from the cache. However, searching the graph takes time, and it could even double the computational overhead for large projects.

Each strategy has a weakness. "speed" is memory-hungry, "memory" wastes time reloading targets from storage, and "lookahead" wastes time traversing the entire dependency graph on every make(). For a better compromise and more control, see the examples in the help file of drake_envir().

layout — config$layout, where config is the return value from a prior call to drake_config(). If your plan or environment have changed since the last make(), do not supply a layout argument. Otherwise, supplying one could save time.

lock_envir — Logical, whether to lock config$envir during make(). If TRUE, make() quits in error whenever a command in your drake plan (or prework) tries to add, remove, or modify non-hidden variables in your environment/workspace/R session. This is extremely important for ensuring the purity of your functions and the reproducibility/credibility/trust you can place in your project. lock_envir will be set to a default of TRUE in drake version 7.0.0 and higher.

Value

nothing

Interactive mode

In interactive sessions, consider r_make(), r_outdated(), etc. rather than make(), outdated(), etc. The r_*() drake functions are more reproducible when the session is interactive.

A serious drake workflow should be consistent and reliable, ideally with the help of a master R script. This script should begin in a fresh R session, load your packages and functions in a dependable manner, and then run make(). Example: https://github.com/wlandau/drake-examples/tree/master/gsp. Batch mode, especially within a container, is particularly helpful.

Interactive R sessions are still useful, but they easily grow stale. Targets can falsely invalidate if you accidentally change a function or data object in your environment. So in interactive mode, make() now pauses with a menu to protect you from environment-related instability.

You can control this menu with the drake_make_menu global option. Run options(drake_make_menu = TRUE) to show the menu once per session and options(drake_make_menu = FALSE) to disable it entirely. You may wish to add a call to options() in your local .Rprofile file.

Self-invalidation

It is possible to construct a workflow that tries to invalidate itself. Example:

```
plan <- drake_plan(
  x = {
    data(mtcars)
    mtcars$mpg
```

```
  },
  y = mean(x)
)
```

In this plan, the very act of building `x` changes the dependencies of `x`. In other words, without safeguards, `x` would not be up to date at the end of `make(plan)`. Please try to avoid workflows that modify the global environment. Otherwise, `make(plan)` will throw an error. To avoid this error, you can run `make(plan, lock_envir = FALSE)`. There are legitimate use cases for `lock_envir = FALSE` (example: https://ropenscilabs.github.io/drake-manual/hpc.html#parallel-computing-within-targets) # nolint but most workflows should stick with the default `lock_envir = TRUE`.

See Also

`drake_plan()`, `drake_config()`, `vis_drake_graph()`, `outdated()`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
config <- drake_config(my_plan)
outdated(config) # Which targets need to be (re)built?
make(my_plan, jobs = 2) # Build what needs to be built.
outdated(config) # Everything is up to date.
# Change one of your imported function dependencies.
reg2 = function(d) {
  d$x3 = d$x^3
  lm(y ~ x3, data = d)
}
outdated(config) # Some targets depend on reg2().
make(my_plan) # Rebuild just the outdated targets.
outdated(config) # Everything is up to date again.
if (requireNamespace("visNetwork", quietly = TRUE)) {
vis_drake_graph(config) # See how they fit in an interactive graph.
make(my_plan, cache_log_file = TRUE) # Write a text log file this time.
vis_drake_graph(config) # The colors changed in the graph.
}
clean() # Start from scratch next time around.
}
})

## End(Not run)
```

`map_plan` *Create a plan that maps a function to a grid of arguments.*

Description

map_plan() is no longer recommended. Consider using transformations instead. Visit https://ropenscilabs.github.io/drake-manual/plans.html#large-plans for the details.

Usage

```
map_plan(args, fun, id = "id", character_only = FALSE, trace = FALSE)
```

Arguments

args	A data frame (or better yet, a tibble) of function arguments to fun. Here, the column names should be the names of the arguments of fun, and each row of args corresponds to a call to fun.
fun	Name of a function to apply the arguments row-by-row. Supply a symbol if character_only is FALSE and a character scalar otherwise.
id	Name of an optional column in args giving the names of the targets. If not supplied, target names will be generated automatically. id should be a symbol if character_only is FALSE and a character scalar otherwise.
character_only	Logical, whether to interpret the fun and id arguments as character scalars or symbols.
trace	Logical, whether to append the columns of args to the output workflow plan data frame. The added columns help "trace back" the original settings that went into building each target. Similar to the trace argument of drake_plan().

Details

map_plan() is like base::Map(): it takes a function name and a grid of arguments, and writes out all the commands calls to apply the function to each row of arguments.

Value

A workflow plan data frame.

See Also

drake_plan()

Examples

```
# For the full tutorial, visit
# https://ropenscilabs.github.io/drake-manual/plans.html#map_plan.
my_model_fit <- function(x1, x2, data) {
  lm(as.formula(paste("mpg ~", x1, "+", x1)), data = data)
}
covariates <- setdiff(colnames(mtcars), "mpg")
args <- t(combn(covariates, 2))
colnames(args) <- c("x1", "x2")
# Use tibble::as_tibble(args) for better printing # nolint
# Below, stringsAsFactors = FALSE is very important! # nolint
```

```
args <- as.data.frame(args, stringsAsFactors = FALSE)
args$data <- "mtcars"
args$data <- rlang::syms(args$data)
args$id <- paste0("fit_", args$x1, "_", args$x2)
# print(args) # Requires `args` to be a tibble # nolint
plan <- map_plan(args, my_model_fit)
plan
# Consider `trace = TRUE` to include the columns in `args`
# in your plan.
plan <- map_plan(args, my_model_fit, trace = TRUE)
# print(plan) # If you have tibble installed # nolint
# And of course, you can execute the plan and
# inspect your results.
cache <- storr::storr_environment()
make(plan, verbose = FALSE, cache = cache)
readd(fit_cyl_disp, cache = cache)
```

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

Description

Checks your workspace/environment and file system.

Usage

```
missed(config)
```

Arguments

`config`	Internal runtime parameter list produced by both `drake_config()` and `make()`.

Value

Character vector of names of missing objects and files.

See Also

`outdated()`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
config <- drake_config(my_plan)
missed(config) # All the imported files and objects should be present.
rm(reg1) # Remove an import dependency from you workspace.
```

```
missed(config) # Should report that reg1 is missing.
}
})

## End(Not run)
```

---

new_cache *Make a new* drake *cache.*

Description

Uses the storr_rds() function from the storr package.

Usage

```
new_cache(path = NULL, verbose = 1L, type = NULL,
  hash_algorithm = NULL, short_hash_algo = NULL,
  long_hash_algo = NULL, ..., console_log_file = NULL)
```

Arguments

path	File path to the cache if the cache is a file system cache.
verbose	Logical or numeric, control printing to the console. • 0 or FALSE: print nothing. • 1 or TRUE: print only targets to build. • 2: plus checks and cache info. • 3: plus missing imports. • 4: plus all imports. • 5: plus execution and total build times for targets. • 6: plus notifications when targets are being stored.
type	Deprecated argument. Once stood for cache type. Use storr to customize your caches instead.
hash_algorithm	Name of a hash algorithm to use. See the algo argument of the digest package for your options.
short_hash_algo	Deprecated on 2018-12-12. Use hash_algorithm instead.
long_hash_algo	Deprecated on 2018-12-12. Use hash_algorithm instead.
...	other arguments to the cache constructor.
console_log_file	Character scalar, connection object (such as stdout()) or NULL. If NULL, console output will be printed to the R console using message(). If a character scalar, console_log_file should be the name of a flat file, and console output will be appended to that file. If a connection object (e.g. stdout()) warnings and messages will be sent to the connection. For example, if console_log_file is stdout(), warnings and messages are printed to the console in real time (in addition to the usual in-bulk printing after each target finishes).

Value

A newly created drake cache as a storr object.

See Also

make()

Examples

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

## End(Not run)
```

---

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

Description

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

Usage

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

Arguments

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

Details

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

Value

Character vector of the names of outdated targets.

See Also

drake_config(), missed(), drake_plan(), make()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
# Recopute the config list early and often to have the
# most current information. Do not modify the config list by hand.
config <- drake_config(my_plan)
outdated(config = config) # Which targets are out of date?
make(my_plan) # Run the projects, build the targets.
config <- drake_config(my_plan)
# Now, everything should be up to date (no targets listed).
outdated(config = config)
# outdated() is sensitive to triggers.
# See the debugging guide: https://ropenscilabs.github.io/drake-manual/debug.html # nolint
config$trigger <- "always"
outdated(config = config)
}
})

## End(Not run)
```

---

plan_to_code *Turn a* drake *workflow plan data frame into a plain R script file.*

Description

code_to_plan(), plan_to_code(), and plan_to_notebook() together illustrate the relationships between drake plans, R scripts, and R Markdown documents. In the file generated by plan_to_code(), every target/command pair becomes a chunk of code. Targets are arranged in topological order so dependencies are available before their downstream targets. Please note:

1. You are still responsible for loading your project's packages, imported functions, etc.
2. Triggers disappear.

Usage

```
plan_to_code(plan, con = stdout())
```

Arguments

plan	Workflow plan data frame. See drake_plan() for details.
con	A file path or connection to write to.

See Also

drake_plan(), make(), code_to_plan(), plan_to_notebook()

Examples

```
plan <- drake_plan(
  raw_data = read_excel(file_in("raw_data.xlsx")),
  data = raw_data,
  hist = create_plot(data),
  fit = lm(Sepal.Width ~ Petal.Width + Species, data)
)
file <- tempfile()
# Turn the plan into an R script a the given file path.
plan_to_code(plan, file)
# Here is what the script looks like.
cat(readLines(file), sep = "\n")
# Convert back to a drake plan.
if (requireNamespace("CodeDepends")) {
  code_to_plan(file)
}
```

---

plan_to_notebook *Turn a* drake *workflow plan data frame into an R notebook,*

Description

code_to_plan(), plan_to_code(), and plan_to_notebook() together illustrate the relationships between drake plans, R scripts, and R Markdown documents. In the file generated by plan_to_code(), every target/command pair becomes a chunk of code. Targets are arranged in topological order so dependencies are available before their downstream targets. Please note:

1. You are still responsible for loading your project's packages, imported functions, etc.
2. Triggers disappear.

Usage

```
plan_to_notebook(plan, con)
```

Arguments

plan	Workflow plan data frame. See drake_plan() for details.
con	A file path or connection to write to.

See Also

drake_plan(), make(), code_to_plan(), plan_to_code()

Examples

```
if (requireNamespace("CodeDepends")) {
if (suppressWarnings(require("knitr"))) {
plan <- drake_plan(
  raw_data = read_excel(file_in("raw_data.xlsx")),
  data = raw_data,
  hist = create_plot(data),
  fit = lm(Sepal.Width ~ Petal.Width + Species, data)
)
file <- tempfile()
# Turn the plan into an R notebook a the given file path.
plan_to_notebook(plan, file)
# Here is what the script looks like.
cat(readLines(file), sep = "\n")
# Convert back to a drake plan.
code_to_plan(file)
}
}
```

---

predict_runtime *Predict the elapsed runtime of the next call to* make() *for non-staged parallel backends.*

Description

Take the past recorded runtimes times from build_times() and use them to predict how the targets will be distributed among the available workers in the next make(). Then, predict the overall runtime to be the runtime of the slowest (busiest) workers. Predictions only include the time it takes to run the targets, not overhead/preprocessing from drake itself.

Usage

```
predict_runtime(config, targets = NULL, from_scratch = FALSE,
  targets_only = NULL, jobs = 1, known_times = numeric(0),
  default_time = 0, warn = TRUE)
```

Arguments

config	Optional internal runtime parameter list of produced by both make() and drake_config().
targets	Character vector, names of targets. Predict the runtime of building these targets plus dependencies. Defaults to all targets.
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	Deprecated.
jobs	The jobs argument of your next planned make(). How many targets to do you plan to have running simultaneously?
known_times	A named numeric vector with targets/imports as names and values as hypothetical runtimes in seconds. Use this argument to overwrite any of the existing build times or the default_time.
default_time	Number of seconds to assume for any target or import with no recorded runtime (from build_times()) or anything in known_times.
warn	Logical, whether to warn the user about any targets with no available runtime, either in known_times or build_times(). The times for these targets default to default_time.

Value

Predicted total runtime of the next call to make().

See Also

predict_workers(), build_times(), make()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Run the project, build the targets.
config <- drake_config(my_plan)
known_times <- rep(7200, nrow(my_plan))
names(known_times) <- my_plan$target
known_times
# Predict the runtime
if (requireNamespace("lubridate", quietly = TRUE)) {
predict_runtime(
  config,
  jobs = 7,
  from_scratch = TRUE,
  known_times = known_times
)
predict_runtime(
  config,
  jobs = 8,
  from_scratch = TRUE,
  known_times = known_times
)
balance <- predict_workers(
  config,
  jobs = 7,
  from_scratch = TRUE,
  known_times = known_times
```

```
)
balance
}
}
})

## End(Not run)
```

`predict_workers` *Predict the load balancing of the next call to* `make()` *for non-staged parallel backends.*

Description

Take the past recorded runtimes times from `build_times()` and use them to predict how the targets will be distributed among the available workers in the next `make()`. Predictions only include the time it takes to run the targets, not overhead/preprocessing from `drake` itself.

Usage

```
predict_workers(config, targets = NULL, from_scratch = FALSE,
  targets_only = NULL, jobs = 1, known_times = numeric(0),
  default_time = 0, warn = TRUE)
```

Arguments

`config`	Optional internal runtime parameter list of produced by both `make()` and `drake_config()`.
`targets`	Character vector, names of targets. Predict the runtime of building these targets plus dependencies. Defaults to all targets.
`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`	Deprecated.
`jobs`	The `jobs` argument of your next planned `make()`. How many targets to do you plan to have running simultaneously?
`known_times`	A named numeric vector with targets/imports as names and values as hypothetical runtimes in seconds. Use this argument to overwrite any of the existing build times or the `default_time`.
`default_time`	Number of seconds to assume for any target or import with no recorded runtime (from `build_times()`) or anything in `known_times`.
`warn`	Logical, whether to warn the user about any targets with no available runtime, either in `known_times` or `build_times()`. The times for these targets default to `default_time`.

Value

A data frame showing one likely arrangement of targets assigned to parallel workers.

See Also

predict_runtime(), build_times(), make()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Run the project, build the targets.
config <- drake_config(my_plan)
known_times <- rep(7200, nrow(my_plan))
names(known_times) <- my_plan$target
known_times
# Predict the runtime
if (requireNamespace("lubridate", quietly = TRUE)) {
predict_runtime(
  config = config,
  jobs = 7,
  from_scratch = TRUE,
  known_times = known_times
)
predict_runtime(
  config,
  jobs = 8,
  from_scratch = TRUE,
  known_times = known_times
)
balance <- predict_workers(
  config,
  jobs = 7,
  from_scratch = TRUE,
  known_times = known_times
)
balance
}
}
})

## End(Not run)
```

---

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

Description

Objects that drake imported, built, or attempted to build are listed as "done" or "running". Skipped objects are not listed.

Usage

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

Arguments

`...`	Objects to load from the cache, as names (unquoted) or character strings (quoted). Similar to `...` in `remove()` and `rm()`.
`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).
`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project. Ignored if a `cache` is supplied.
`search`	Logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a `cache` is supplied.
`cache`	drake cache. See `new_cache()`. If supplied, `path` and `search` are ignored.
`verbose`	Logical or numeric, control printing to the console. • `0` or `FALSE`: print nothing. • `1` or `TRUE`: print only targets to build. • `2`: plus checks and cache info. • `3`: plus missing imports. • `4`: plus all imports. • `5`: plus execution and total build times for targets. • `6`: plus notifications when targets are being stored.
`jobs`	Number of jobs/workers for parallel processing.
`progress`	Character vector for filtering the build progress results. Defaults to `NULL` (no filtering) to report progress of all objects. Supported filters are `"done"`, `"running"`, `"failed"` and `"none"`.

Value

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

See Also

`diagnose()`, `drake_get_session_info()`, `cached()`, `readd()`, `drake_plan()`, `make()`

Examples

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

## End(Not run)
```

---

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

Description

`readd()` returns an object from the cache, and `loadd()` loads one or more objects from the cache into your environment or session. These objects are usually targets built by `make()`.

Usage

```
readd(target, character_only = FALSE, path = getwd(), search = TRUE,
  cache = drake::get_cache(path = path, search = search, verbose =
  verbose), namespace = NULL, verbose = 1L, show_source = FALSE)

loadd(..., list = character(0), imported_only = NULL, path = getwd(),
  search = TRUE, cache = drake::get_cache(path = path, search = search,
  verbose = verbose), namespace = NULL, envir = parent.frame(),
  jobs = 1, verbose = 1L, deps = FALSE, lazy = "eager",
  graph = NULL, replace = TRUE, show_source = FALSE,
  tidyselect = TRUE, config = NULL)
```

Arguments

`target`	If `character_only` is `TRUE`, then `target` is a character string naming the object to read. Otherwise, `target` is an unquoted symbol with the name of the 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. Ignored if a `cache` is supplied.
`search`	Logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a `cache` is supplied.

cache	drake cache. See new_cache(). If supplied, path and search are ignored.
namespace	Optional character string, name of the storr namespace to read from.
verbose	Logical or numeric, control printing to the console. • 0 or FALSE: print nothing. • 1 or TRUE: print only targets to build. • 2: plus checks and cache info. • 3: plus missing imports. • 4: plus all imports. • 5: plus execution and total build times for targets. • 6: plus notifications when targets are being stored.
show_source	Logical, option to show the command that produced the target or indicate that the object was imported (using show_source()).
...	Targets to load from the cache: as names (symbols) or character strings. If the tidyselect package is installed, you can also supply dplyr-style tidyselect commands such as starts_with(), ends_with(), and one_of().
list	Character vector naming targets to be loaded from the cache. Similar to the list argument of remove().
imported_only	Logical, deprecated.
envir	Environment to load objects into. Defaults to the calling environment (current workspace).
jobs	Number of parallel jobs for loading objects. On non-Windows systems, the loading process for multiple objects can be lightly parallelized via parallel::mclapply(). just set jobs to be an integer greater than 1. On Windows, jobs is automatically demoted to 1.
deps	Logical, whether to load any cached dependencies of the targets instead of the targets themselves. This is useful if you know your target failed and you want to debug the command in an interactive session with the dependencies in your workspace. One caveat: to find the dependencies, loadd() uses information that was stored in a drake_config() list and cached during the last make(). That means you need to have already called make() if you set deps to TRUE.
lazy	Either a string or a logical. Choices: • "eager": no lazy loading. The target is loaded right away with assign(). • "promise": lazy loading with delayedAssign() • "bind": lazy loading with active bindings: bindr::populate_env(). • TRUE: same as "promise". • FALSE: same as "eager".
graph	Deprecated.
replace	Logical. If FALSE, items already in your environment will not be replaced.
tidyselect	Logical, whether to enable tidyselect expressions in ... like starts_with("prefix") and ends_with("suffix").
config	Optional drake_config() object. You should supply one if deps is TRUE.

Details

There are two uses for the loadd() and readd() functions:

1. Exploring the results outside the drake/make() pipeline. When you call make() to run your project, drake puts the targets in a cache, usually a folder called .drake. You may want to inspect the targets afterwards, possibly in an interactive R session. However, the files in the .drake folder are organized in a special format created by the storr package, which is not exactly human-readable. To retrieve a target for manual viewing, use readd(). To load one or more targets into your session, use loadd().
2. In knitr / R Markdown reports. You can borrow drake targets in your active code chunks if you have the right calls to loadd() and readd(). These reports can either run outside the drake pipeline, or better yet, as part of the pipeline itself. If you call knitr_in("your_report.Rmd") inside a drake_plan() command, then make() will scan "your_report.Rmd" for calls to loadd() and readd() in active code chunks, and then treat those loaded targets as dependencies. That way, make() will automatically (re)run the report if those dependencies change.

Value

The cached value of the target.

Note

Please do not put calls to loadd() or readd() inside your custom (imported) functions or the commands in your drake_plan(). This creates confusion inside make(), which has its own ways of interacting with the cache.

See Also

cached(), drake_plan(), make()

cached(), drake_plan(), make()

Examples

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

## End(Not run)
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
```

```
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Run the projects, build the targets.
config <- drake_config(my_plan)
loadd(small) # Load target 'small' into your workspace.
small
# For many targets, you can parallelize loadd()
# using the 'jobs' argument.
loadd(list = c("small", "large"), jobs = 2)
ls()
# Load the dependencies of the target, coef_regression2_small
loadd(coef_regression2_small, deps = TRUE, config = config)
ls()
# Load all the targets listed in the workflow plan
# of the previous `make()`.
# If you do not supply any target names, `loadd()` loads all the targets.
# Be sure your computer has enough memory.
loadd()
ls()
}
})

## End(Not run)
```

`read_drake_seed` *Read the pseudo-random number generator seed of the project.*

Description

When a project is created with `make()` or `drake_config()`, the project's pseudo-random number generator seed is cached. Then, unless the cache is destroyed, the seeds of all the targets will deterministically depend on this one central seed. That way, reproducibility is protected, even under randomness.

Usage

```
read_drake_seed(path = getwd(), search = TRUE, cache = NULL,
  verbose = 1L)
```

Arguments

`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project. Ignored if a `cache` is supplied.
`search`	Logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a `cache` is supplied.
`cache`	drake cache. See `new_cache()`. If supplied, `path` and `search` are ignored.
`verbose`	Logical or numeric, control printing to the console. • `0` or `FALSE`: print nothing.

- 1 or TRUE: print only targets to build.
- 2: plus checks and cache info.
- 3: plus missing imports.
- 4: plus all imports.
- 5: plus execution and total build times for targets.
- 6: plus notifications when targets are being stored.

Value

An integer vector.

Examples

```
cache <- storr::storr_environment() # Just for the examples.
my_plan <- drake_plan(
  target1 = sqrt(1234),
  target2 = sample.int(n = 12, size = 1) + target1
)
tmp <- sample.int(1) # Needed to get a .Random.seed, but not for drake.
digest::digest(.Random.seed) # Fingerprint of the current R session's seed.
make(my_plan, cache = cache) # Run the project, build the targets.
digest::digest(.Random.seed) # Your session's seed did not change.
# drake uses a hard-coded seed if you do not supply one.
read_drake_seed(cache = cache)
readd(target2, cache = cache) # Randomly-generated target data.
clean(target2, cache = cache) # Oops, I removed the data!
tmp <- sample.int(1) # Maybe the R session's seed also changed.
make(my_plan, cache = cache) # Rebuild target2.
# Same as before:
read_drake_seed(cache = cache)
readd(target2, cache = cache)
# You can also supply a seed.
# If your project already exists, it must agree with the project's
# preexisting seed (default: 0)
clean(target2, cache = cache)
make(my_plan, cache = cache, seed = 0)
read_drake_seed(cache = cache)
readd(target2, cache = cache)
# If you want to supply a different seed than 0,
# you need to destroy the cache and start over first.
clean(destroy = TRUE, cache = cache)
cache <- storr::storr_environment() # Just for the examples.
make(my_plan, cache = cache, seed = 1234)
read_drake_seed(cache = cache)
readd(target2, cache = cache)
```

reduce_by *Reduce multiple groupings of targets*

Description

`reduce_by()` are no longer recommended. Consider using transformations instead. Visit https://ropenscilabs.github.io/drake-manual/plans.html#large-plans for the details.

Usage

```
reduce_by(plan, ..., prefix = "target", begin = "", op = " + ",
  end = "", pairwise = TRUE, append = TRUE, filter = NULL,
  sep = "_")
```

Arguments

`plan`	Workflow plan data frame of prespecified targets.
`...`	Symbols, columns of `plan` to define target groupings. A `reduce_plan()` call is applied for each grouping. Groupings with all `NA`s in the selector variables are ignored.
`prefix`	Character, prefix for naming the new targets. Suffixes are generated from the values of the columns specified in `...`.
`begin`	Character, code to place at the beginning of each step in the reduction.
`op`	Binary operator to apply in the reduction
`end`	Character, code to place at the end of each step in the reduction.
`pairwise`	Logical, whether to create multiple new targets, one for each pair/step in the reduction (`TRUE`), or to do the reduction all in one command.
`append`	Logical. If `TRUE`, the output will include the original rows in the `plan` argument. If `FALSE`, the output will only include the new targets and commands.
`filter`	An expression like you would pass to `dplyr::filter()`. The rows for which `filter` evaluates to `TRUE` will be gathered, and the rest will be excluded from gathering. Why not just call `dplyr::filter()` before `gather_by()`? Because `gather_by(append = TRUE, filter = my_column == "my_value")` gathers on some targets while including all the original targets in the output. See the examples for a demonstration.
`sep`	Character scalar, delimiter for creating the names of new targets.

Details

Perform several calls to `reduce_plan()` based on groupings from columns in the plan, and then row-bind the new targets to the plan.

Value

A workflow plan data frame.

See Also

drake_plan()

Examples

```
plan <- drake_plan(
  data = get_data(),
  informal_look = inspect_data(data, mu = mu__),
  bayes_model = bayesian_model_fit(data, prior_mu = mu__)
)
plan <- evaluate_plan(plan, rules = list(mu__ = 1:2), trace = TRUE)
plan
reduce_by(plan, mu___from, begin = "list(", end = ")", op = ", ")
reduce_by(plan, mu__)
reduce_by(plan, mu__, append = TRUE)
reduce_by(plan, mu__, append = FALSE)
reduce_by(plan) # Reduce all the targets.
reduce_by(plan, append = FALSE)
reduce_by(plan, pairwise = FALSE)
# You can filter out the informal_look_* targets beforehand
# if you only want the bayes_model_* ones to be reduced.
# The advantage here is that if you also need `append = TRUE`,
# only the bayes_model_* targets will be reduced, but
# the informal_look_* targets will still be included
# in the output.
reduce_by(
  plan,
  mu___from,
  append = TRUE,
  filter = mu___from == "bayes_model"
)
```

---

reduce_plan *Write commands to reduce several targets down to one.*

Description

reduce_plan() is no longer recommended. Consider using transformations instead. Visit https://ropenscilabs.github.io/drake-manual/plans.html#large-plans for the details.

Usage

```
reduce_plan(plan = NULL, target = "target", begin = "", op = " + ",
  end = "", pairwise = TRUE, append = FALSE, sep = "_")
```

Arguments

plan	Workflow plan data frame of prespecified targets.
target	Name of the new reduced target.
begin	Character, code to place at the beginning of each step in the reduction.
op	Binary operator to apply in the reduction
end	Character, code to place at the end of each step in the reduction.
pairwise	Logical, whether to create multiple new targets, one for each pair/step in the reduction (TRUE), or to do the reduction all in one command.
append	Logical. If TRUE, the output will include the original rows in the plan argument. If FALSE, the output will only include the new targets and commands.
sep	Character scalar, delimiter for creating new target names.

Details

Creates a new workflow plan data frame with the commands to do a reduction (i.e. to repeatedly apply a binary operator to pairs of targets to produce one target).

Value

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

See Also

drake_plan()

Examples

```
# Workflow plan for datasets:
x_plan <- evaluate_plan(
  drake_plan(x = VALUE),
  wildcard = "VALUE",
  values = 1:8
)
x_plan
# Create a new target from the sum of the others.
reduce_plan(x_plan, target = "x_sum", pairwise = FALSE, append = FALSE)
# Optionally include the original rows with `append = TRUE`.
reduce_plan(x_plan, target = "x_sum", pairwise = FALSE, append = TRUE)
# For memory efficiency and parallel computing,
# reduce pairwise:
reduce_plan(x_plan, target = "x_sum", pairwise = TRUE, append = FALSE)
# Optionally define your own function and use it as the
# binary operator in the reduction.
x_plan <- evaluate_plan(
  drake_plan(x = VALUE),
  wildcard = "VALUE",
  values = 1:9
```

```
)
x_plan
reduce_plan(
  x_plan, target = "x_sum", pairwise = TRUE,
  begin = "fun(", op = ", ", end = ")"
)
```

---

render_drake_ggraph *Render a static* ggplot2/ggraph *visualization from* drake_graph_info() *output.*

Description

This function requires packages ggplot2 and ggraph. Install them with `install.packages(c("ggplot2", "ggraph"))`.

Usage

```
render_drake_ggraph(graph_info, main = graph_info$default_title)
```

Arguments

graph_info	List of data frames generated by drake_graph_info(). There should be 3 data frames: nodes, edges, and legend_nodes.
main	Character string, title of the graph.

Value

A ggplot2 object, which you can modify with more layers, show with plot(), or save as a file with ggsave().

See Also

vis_drake_graph(), sankey_drake_graph(), drake_ggraph()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
if (requireNamespace("ggraph", quietly = TRUE)) {
  # Instead of jumpting right to vis_drake_graph(), get the data frames
  # of nodes, edges, and legend nodes.
  config <- drake_config(my_plan) # Internal configuration list
  drake_ggraph(config) # Jump straight to the static graph.
  # Get the node and edge info that vis_drake_graph() just plotted:
  graph <- drake_graph_info(config)
  render_drake_ggraph(graph)
}
})
```

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

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

Description

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

Usage

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

Arguments

`graph_info`	List of data frames generated by `drake_graph_info()`. There should be 3 data frames: `nodes`, `edges`, and `legend_nodes`.
`file`	Name of a file to save the graph. If `NULL` or `character(0)`, no file is saved and the graph is rendered and displayed within R. If the file ends in a `.png`, `.jpg`, `.jpeg`, or `.pdf` extension, then a static image will be saved. In this case, the webshot package and PhantomJS are required: `install.packages("webshot"); webshot::install_p` If the file does not end in a `.png`, `.jpg`, `.jpeg`, or `.pdf` extension, an HTML file will be saved, and you can open the interactive graph using a web browser.
`layout`	Name of an igraph layout to use, such as 'layout_with_sugiyama' or 'layout_as_tree'. Be careful with 'layout_as_tree': the graph is a directed acyclic graph, but not necessarily a tree.
`direction`	An argument to `visNetwork::visHierarchicalLayout()` indicating the direction of the graph. Options include 'LR', 'RL', 'DU', and 'UD'. At the time of writing this, the letters must be capitalized, but this may not always be the case ;) in the future.
`hover`	Logical, whether to show the command that generated the target when you hover over a node with the mouse. For imports, the label does not change with hovering.
`main`	Character string, 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. The `selfcontained` argument only applies to HTML files. In other words, if `file` is a PNG, PDF, or JPEG file, for instance, the point is moot.
`navigationButtons`	Logical, whether to add navigation buttons with `visNetwork::visInteraction(navigationButtons =`

ncol_legend	Number of columns in the legend nodes. To remove the legend entirely, set ncol_legend to NULL or 0.
collapse	Logical, whether to allow nodes to collapse if you double click on them. Analogous to visNetwork::visOptions(collapse = TRUE) or visNetwork::visOptions(collapse = TRU
...	Arguments passed to visNetwork().

Value

A visNetwork graph.

See Also

vis_drake_graph(), sankey_drake_graph(), drake_ggraph()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
if (requireNamespace("visNetwork", quietly = TRUE)) {
# Instead of jumpting right to vis_drake_graph(), get the data frames
# of nodes, edges, and legend nodes.
config <- drake_config(my_plan) # Internal configuration list
vis_drake_graph(config) # Jump straight to the interactive graph.
# Get the node and edge info that vis_drake_graph() just plotted:
graph <- drake_graph_info(config)
# You can pass the data frames right to render_drake_graph()
# (as in vis_drake_graph()) or you can create
# your own custom visNewtork graph.
render_drake_graph(graph, width = '100%') # Width is passed to visNetwork.
# Optionally visualize clusters.
config$plan$large_data <- grepl("large", config$plan$target)
graph <- drake_graph_info(
  config, group = "large_data", clusters = c(TRUE, FALSE))
render_drake_graph(graph)
# You can even use clusters given to you for free in the `graph$nodes`
# data frame.
graph <- drake_graph_info(
  config, group = "status", clusters = "imported")
render_drake_graph(graph)
}
}
})

## End(Not run)
```

render_sankey_drake_graph *Render a Sankey diagram from* drake_graph_info().

Description

This function is called inside sankey_drake_graph(), which typical users call more often. A legend is unfortunately unavailable for the graph itself (https://github.com/christophergandrud/networkD3/issues/240) but you can see what all the colors mean with visNetwork::visNetwork(drake::legend_nodes(

Usage

```
render_sankey_drake_graph(graph_info, file = character(0),
  selfcontained = FALSE, ...)
```

Arguments

graph_info	List of data frames generated by drake_graph_info(). There should be 3 data frames: nodes, edges, and legend_nodes.
file	Name of a file to save the graph. If NULL or character(0), no file is saved and the graph is rendered and displayed within R. If the file ends in a .png, .jpg, .jpeg, or .pdf extension, then a static image will be saved. In this case, the webshot package and PhantomJS are required: install.packages("webshot"); webshot::install_p If the file does not end in a .png, .jpg, .jpeg, or .pdf extension, an HTML file will be saved, and you can open the interactive graph using a web browser.
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 networkD3::sankeyNetwork().

Value

A visNetwork graph.

See Also

sankey_drake_graph(), vis_drake_graph(), drake_ggraph()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
if (suppressWarnings(require("knitr"))) {
if (requireNamespace("networkD3", quietly = TRUE)) {
if (requireNamespace("visNetwork", quietly = TRUE)) {
# Instead of jumpting right to sankey_drake_graph(), get the data frames
```

```
# of nodes, edges, and legend nodes.
config <- drake_config(my_plan) # Internal configuration list
sankey_drake_graph(config) # Jump straight to the interactive graph.
# Show the legend separately.
visNetwork::visNetwork(nodes = drake::legend_nodes())
# Get the node and edge info that sankey_drake_graph() just plotted:
graph <- drake_graph_info(config)
# You can pass the data frames right to render_sankey_drake_graph()
# (as in sankey_drake_graph()) or you can create
# your own custom visNewtork graph.
render_sankey_drake_graph(graph, width = '100%') # Width is passed to visNetwork.
# Optionally visualize clusters.
config$plan$large_data <- grepl("large", config$plan$target)
graph <- drake_graph_info(
config, group = "large_data", clusters = c(TRUE, FALSE))
render_sankey_drake_graph(graph)
# You can even use clusters given to you for free in the `graph$nodes`
# data frame.
graph <- drake_graph_info(config, group = "status", clusters = "imported")
render_sankey_drake_graph(graph)
}
}
}
})

## End(Not run)
```

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

Description

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

Usage

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

Arguments

`targets`	Character vector, names of the targets to rescue. As with many other drake utility functions, the word `target` is defined generally in this case, encompassing imports as well as true targets. If `targets` is `NULL`, everything in the cache is rescued.

path — Character, either the root file path of a drake project or a folder containing the root (top-level working directory where you plan to call make()). If this is too confusing, feel free to just use storr::storr_rds() to get the cache. If search = FALSE, path must be the root. If search = TRUE, you can specify any subdirectory of the project. Let's say "/home/you/my_project" is the root. The following are equivalent and correct:

- get_cache(path = "/home/you/my_project", search = FALSE)
- get_cache(path = "/home/you/my_project", search = TRUE)
- get_cache(path = "/home/you/my_project/subdir/x", search = TRUE)
- get_cache(path = "/home/you/my_project/.drake", search = TRUE)
- get_cache(path = "/home/you/my_project/.drake/keys", search = TRUE)

search — Logical. If TRUE, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a cache is supplied.

verbose — Logical or numeric, control printing to the console.

- 0 or FALSE: print nothing.
- 1 or TRUE: print only targets to build.
- 2: plus checks and cache info.
- 3: plus missing imports.
- 4: plus all imports.
- 5: plus execution and total build times for targets.
- 6: plus notifications when targets are being stored.

force — Deprecated.

cache — A storr cache object.

jobs — Number of jobs for light parallelism (disabled on Windows).

garbage_collection — Logical, whether to do garbage collection as a final step. See drake_gc() and clean() for details.

Value

The rescued drake/storr cache.

See Also

get_cache(), cached(), drake_gc(), clean()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Run the project, build targets. This creates the cache.
# Remove dangling cache files that could cause errors.
rescue_cache(jobs = 2)
# Alternatively, just rescue targets 'small' and 'large'.
```

```
# Rescuing specific targets is usually faster.
rescue_cache(targets = c("small", "large"))
}
})

## End(Not run)
```

`running` *List running targets.*

Description

List the targets that either (1) are currently being built during a call to `make()`, or (2) if `make()` was interrupted, the targets that were running at the time.

Usage

```
running(path = getwd(), search = TRUE, cache = drake::get_cache(path
  = path, search = search, verbose = verbose), verbose = 1L)
```

Arguments

`path`	Root directory of the drake project, or if `search` is `TRUE`, either the project root or a subdirectory of the project. Ignored if a `cache` is supplied.
`search`	Logical. If `TRUE`, search parent directories to find the nearest drake cache. Otherwise, look in the current working directory only. Ignored if a `cache` is supplied.
`cache`	drake cache. See `new_cache()`. If supplied, `path` and `search` are ignored.
`verbose`	Logical or numeric, control printing to the console. • `0` or `FALSE`: print nothing. • `1` or `TRUE`: print only targets to build. • `2`: plus checks and cache info. • `3`: plus missing imports. • `4`: plus all imports. • `5`: plus execution and total build times for targets. • `6`: plus notifications when targets are being stored.

Value

A character vector of target names.

See Also

`failed()`, `make()`

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Run the project, build the targets.
running() # Everything should be done.
# nolint start
# Run make() in one R session...
# slow_plan <- drake_plan(x = Sys.sleep(2))
# make(slow_plan)
# and see the progress in another session.
# running()
# nolint end
}
})

## End(Not run)
```

---

r_make *Experimental: reproducible R session management for drake functions*

Description

A word of caution: r_make() and friends are still new and experimental.

drake searches your environment to detect dependencies, so functions like make(), outdated(), etc. are designed to run in fresh clean R sessions. Wrappers r_make(), r_outdated(), etc. run reproducibly even if your current R session is old and stale.

Usage

```
r_make(source = NULL, r_fn = NULL, r_args = list())

r_drake_build(..., source = NULL, r_fn = NULL, r_args = list())

r_outdated(..., source = NULL, r_fn = NULL, r_args = list())

r_missed(..., source = NULL, r_fn = NULL, r_args = list())

r_drake_graph_info(..., source = NULL, r_fn = NULL, r_args = list())

r_vis_drake_graph(..., source = NULL, r_fn = NULL, r_args = list())

r_sankey_drake_graph(..., source = NULL, r_fn = NULL,
  r_args = list())

r_drake_ggraph(..., source = NULL, r_fn = NULL, r_args = list())
```

```
r_predict_runtime(..., source = NULL, r_fn = NULL, r_args = list())

r_predict_workers(..., source = NULL, r_fn = NULL, r_args = list())
```

Arguments

source	Path to an R script file that loads packages, functions, etc. and returns a drake_config() object. There are 3 ways to set this path. 1. Pass an explicit file path. 2. Call options(drake_source = "path_to_your_script.R"). 3. Just create a file called "_drake.R" in your working directory and supply nothing to source.
r_fn	A callr function such as callr::r or callr::r_bg. Example: r_make(r_fn = callr::r).
r_args	List of arguments to r_fn, not including func or args. Example: r_make(r_fn = callr::r_bg, r_args
...	Arguments to the inner function. For example, if you want to call r_vis_drake_graph(), the inner function is vis_drake_graph(), and selfcontained is an example argument you could supply to the ellipsis.

Details

r_outdated() runs the four steps below. r_make() etc. are similar.

1. Launch a new callr::r() session.
2. In that fresh session, run the R script from the source argument. This script loads packages, functions, global options, etc. and returns a drake_config() object.
3. In that same session, run outdated() with the config argument from step 2.
4. Return the result back to master process (e.g. your interactive R session).

See Also

make()

Examples

```
## Not run:
test_with_dir("quarantine side effects", {
writeLines(
  c(
    "library(drake)",
    "load_mtcars_example()",
    "drake_config(my_plan)"
  ),
  "_drake.R" # default value of the `source` argument
)
cat(readLines("_drake.R"), sep = "\n")
r_outdated()
r_make()
r_outdated()
```

```
})

## End(Not run)
```

---

sankey_drake_graph *Show a Sankey graph of your drake project.*

Description

To save time for repeated plotting, this function is divided into drake_graph_info() and render_sankey_drake_graph(). A legend is unfortunately unavailable for the graph itself (https://github.com/christophergandrud/networkD3/issues/240) but you can see what all the colors mean with visNetwork::visNetwork(drake::legend_nodes(

Usage

```
sankey_drake_graph(config, file = character(0), selfcontained = FALSE,
  build_times = "build", digits = 3, targets_only = FALSE,
  from = NULL, mode = c("out", "in", "all"), order = NULL,
  subset = NULL, make_imports = TRUE, from_scratch = FALSE,
  group = NULL, clusters = NULL, show_output_files = TRUE, ...)
```

Arguments

config	A drake_config() configuration list. You can get one as a return value from make() as well.
file	Name of a file to save the graph. If NULL or character(0), no file is saved and the graph is rendered and displayed within R. If the file ends in a .png, .jpg, .jpeg, or .pdf extension, then a static image will be saved. In this case, the webshot package and PhantomJS are required: install.packages("webshot"); webshot::install_p If the file does not end in a .png, .jpg, .jpeg, or .pdf extension, an HTML file will be saved, and you can open the interactive graph using a web browser.
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	Character string or logical. If character, the choices are 1. "build": runtime of the command plus the time it take to store the target or import. 2. "command": just the runtime of the command. 3. "none": no build times. If logical, build_times selects whether to show the times from 'build_times(..., type = "build")" or use no build times at all. See build_times() for details.
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.
from	Optional collection of target/import names. If from is nonempty, the graph will restrict itself to a neighborhood of from. Control the neighborhood with mode and order.

mode	Which direction to branch out in the graph to create a neighborhood around from. Use "in" to go upstream, "out" to go downstream, and "all" to go both ways and disregard edge direction altogether.
order	How far to branch out to create a neighborhood around from. Defaults to as far as possible. If a target is in the neighborhood, then so are all of its custom file_out() files if show_output_files is TRUE. That means the actual graph order may be slightly greater than you might expect, but this ensures consistency between show_output_files = TRUE and show_output_files = FALSE.
subset	Optional character vector. Subset of targets/imports to display in the graph. Applied after from, mode, and order. Be advised: edges are only kept for adjacent nodes in subset. If you do not select all the intermediate nodes, edges will drop from the graph.
make_imports	Logical, whether to make the imports first. Set to FALSE to increase speed and risk using obsolete information.
from_scratch	Logical, whether to assume all the targets will be made from scratch on the next make(). Makes all targets outdated, but keeps information about build progress in previous make()s.
group	Optional character scalar, name of the column used to group nodes into columns. All the columns names of your config$plan are choices. The other choices (such as "status") are column names in the nodes . To group nodes into clusters in the graph, you must also supply the clusters argument.
clusters	Optional character vector of values to cluster on. These values must be elements of the column of the nodes data frame that you specify in the group argument to drake_graph_info().
show_output_files	Logical, whether to include file_out() files in the graph.
...	Arguments passed to networkD3::sankeyNetwork().

Value

A visNetwork graph.

See Also

render_sankey_drake_graph(), vis_drake_graph(), drake_ggraph()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
config <- drake_config(my_plan)
if (requireNamespace("networkD3", quietly = TRUE)) {
if (requireNamespace("visNetwork", quietly = TRUE)) {
# Plot the network graph representation of the workflow.
sankey_drake_graph(config, width = '100%') # The width is passed to visNetwork
# Show the legend separately.
```

```
visNetwork::visNetwork(nodes = drake::legend_nodes())
make(my_plan) # Run the project, build the targets.
sankey_drake_graph(config) # The black nodes from before are now green.
# Plot a subgraph of the workflow.
sankey_drake_graph(config, from = c("small", "reg2"))
# Optionally visualize clusters.
config$plan$large_data <- grepl("large", config$plan$target)
sankey_drake_graph(
  config, group = "large_data", clusters = c(TRUE, FALSE))
# You can even use clusters given to you for free in the `graph$nodes`
# data frame of `drake_graph_info()`.
sankey_drake_graph(config, group = "status", clusters = "imported")
}
}
}
})

## End(Not run)
```

---

show_source *Show how a target/import was produced.*

Description

Show the command that produced a target or indicate that the object or file was imported.

Usage

```
show_source(target, config, character_only = FALSE)
```

Arguments

target	Symbol denoting the target or import or a character vector if character_only is TRUE.
config	A drake_config() list.
character_only	Logical, whether to interpret target as a symbol (FALSE) or character vector (TRUE).

Examples

```
plan <- drake_plan(x = sample.int(15))
cache <- storr::storr_environment() # custom in-memory cache
make(plan, cache = cache)
config <- drake_config(plan, cache = cache)
show_source(x, config)
```

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

Description

List all the layout in your project's dependency network.

Usage

```
tracked(config)
```

Arguments

config	An output list from drake_config().

Value

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

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Load the canonical example for drake.
# List all the targets/imports that are reproducibly tracked.
config <- drake_config(my_plan)
tracked(config)
}
})

## End(Not run)
```

trigger *Customize the decision rules for rebuilding targets*

Description

Use this function inside a target's command in your drake_plan() or the trigger argument to make() or drake_config(). For details, see the chapter on triggers in the user manual: https://ropenscilabs.github.io/drake-manual

Usage

```
trigger(command = TRUE, depend = TRUE, file = TRUE,
  condition = FALSE, change = NULL, mode = c("whitelist",
  "blacklist", "condition"))
```

Arguments

command	Logical, whether to rebuild the target if the drake_plan() command changes.
depend	Logical, whether to rebuild if a non-file dependency changes.
file	Logical, whether to rebuild the target if a file_in()/file_out()/knitr_in() file changes.
condition	R code (expression or language object) that returns a logical. The target will rebuild if the code evaluates to TRUE.
change	R code (expression or language object) that returns any value. The target will rebuild if that value is different from last time or not already cached.
mode	A character scalar equal to "whitelist" (default) or "blacklist" or "condition". With the mode argument, you can choose how the condition trigger factors into the decision to build or skip the target. Here are the options.

- "whitelist" (default): we *rebuild* the target whenever condition evaluates to TRUE. Otherwise, we defer to the other triggers. This behavior is the same as the decision rule described in the "Details" section of this help file.
- "blacklist": we *skip* the target whenever condition evaluates to FALSE. Otherwise, we defer to the other triggers.
- "condition": here, the condition trigger is the only decider, and we ignore all the other triggers. We *rebuild* target whenever condition evaluates to TRUE and *skip* it whenever condition evaluates to FALSE.

Details

A target always builds if it has not been built before. Triggers allow you to customize the conditions under which a pre-existing target *re*builds. By default, the target will rebuild if and only if:

- Any of command, depend, or file is TRUE, or
- condition evaluates to TRUE, or
- change evaluates to a value different from last time. The above steps correspond to the "whitelist" decision rule. You can select other decision rules with the mode argument described in this help file. On another note, there may be a slight efficiency loss if you set complex triggers for change and/or condition because drake needs to load any required dependencies into memory before evaluating these triggers.

Value

A list of trigger specification details that drake processes internally when it comes time to decide whether to build the target.

See Also

drake_plan(), make()

Examples

```
# A trigger is just a set of decision rules
# to decide whether to build a target.
trigger()
# This trigger will build a target on Tuesdays
# and when the value of an online dataset changes.
trigger(condition = today() == "Tuesday", change = get_online_dataset())
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
# You can use a global trigger argument:
# for example, to always run everything.
make(my_plan, trigger = trigger(condition = TRUE))
make(my_plan, trigger = trigger(condition = TRUE))
# You can also define specific triggers for each target.
plan <- drake_plan(
  x = sample.int(15),
  y = target(
    command = x + 1,
    trigger = trigger(depend = FALSE)
  )
)
# Now, when x changes, y will not.
make(plan)
make(plan)
plan$command[1] <- "sample.int(16)" # change x
make(plan)
}
})

## End(Not run)
```

---

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

Description

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

Usage

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

```
  make_imports = TRUE, from_scratch = FALSE, group = NULL,
  clusters = NULL, show_output_files = TRUE, collapse = TRUE, ...)
```

Arguments

config	A drake_config() configuration list. You can get one as a return value from make() as well.
file	Name of a file to save the graph. If NULL or character(0), no file is saved and the graph is rendered and displayed within R. If the file ends in a .png, .jpg, .jpeg, or .pdf extension, then a static image will be saved. In this case, the webshot package and PhantomJS are required: install.packages("webshot"); webshot::install_p If the file does not end in a .png, .jpg, .jpeg, or .pdf extension, an HTML file will be saved, and you can open the interactive graph using a web browser.
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. The selfcontained argument only applies to HTML files. In other words, if file is a PNG, PDF, or JPEG file, for instance, the point is moot.
build_times	Character string or logical. If character, the choices are 1. "build": runtime of the command plus the time it take to store the target or import. 2. "command": just the runtime of the command. 3. "none": no build times. If logical, build_times selects whether to show the times from 'build_times(..., type = "build")" or use no build times at all. See build_times() for details.
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.
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	Character string, title of the graph.
direction	An argument to visNetwork::visHierarchicalLayout() indicating the direction of the graph. Options include 'LR', 'RL', 'DU', and 'UD'. At the time of writing this, the letters must be capitalized, but this may not always be the case ;) in the future.
hover	Logical, whether to show text (file contents, commands, etc.) when you hover your cursor over a node.
navigationButtons	Logical, whether to add navigation buttons with visNetwork::visInteraction(navigationButtons =
from	Optional collection of target/import names. If from is nonempty, the graph will restrict itself to a neighborhood of from. Control the neighborhood with mode and order.
mode	Which direction to branch out in the graph to create a neighborhood around from. Use "in" to go upstream, "out" to go downstream, and "all" to go both ways and disregard edge direction altogether.

order	How far to branch out to create a neighborhood around from. Defaults to as far as possible. If a target is in the neighborhood, then so are all of its custom file_out() files if show_output_files is TRUE. That means the actual graph order may be slightly greater than you might expect, but this ensures consistency between show_output_files = TRUE and show_output_files = FALSE.
subset	Optional character vector. Subset of targets/imports to display in the graph. Applied after from, mode, and order. Be advised: edges are only kept for adjacent nodes in subset. If you do not select all the intermediate nodes, edges will drop from the graph.
ncol_legend	Number of columns in the legend nodes. To remove the legend entirely, set ncol_legend to NULL or 0.
full_legend	Logical. If TRUE, all the node types are printed in the legend. If FALSE, only the node types used are printed in the legend.
make_imports	Logical, whether to make the imports first. Set to FALSE to increase speed and risk using obsolete information.
from_scratch	Logical, whether to assume all the targets will be made from scratch on the next make(). Makes all targets outdated, but keeps information about build progress in previous make()s.
group	Optional character scalar, name of the column used to group nodes into columns. All the columns names of your config$plan are choices. The other choices (such as "status") are column names in the nodes . To group nodes into clusters in the graph, you must also supply the clusters argument.
clusters	Optional character vector of values to cluster on. These values must be elements of the column of the nodes data frame that you specify in the group argument to drake_graph_info().
show_output_files	Logical, whether to include file_out() files in the graph.
collapse	Logical, whether to allow nodes to collapse if you double click on them. Analogous to visNetwork::visOptions(collapse = TRUE) or visNetwork::visOptions(collapse = TRU
...	Arguments passed to visNetwork().

Value

A visNetwork graph.

See Also

render_drake_graph(), sankey_drake_graph(), drake_ggraph()

Examples

```
## Not run:
test_with_dir("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
config <- drake_config(my_plan)
# Plot the network graph representation of the workflow.
```

```
if (requireNamespace("visNetwork", quietly = TRUE)) {
vis_drake_graph(config, width = '100%') # The width is passed to visNetwork
make(my_plan) # Run the project, build the targets.
vis_drake_graph(config) # The red nodes from before are now green.
# Plot a subgraph of the workflow.
vis_drake_graph(
  config,
  from = c("small", "reg2"),
  to = "summ_regression2_small"
)
# Optionally visualize clusters.
config$plan$large_data <- grepl("large", config$plan$target)
vis_drake_graph(
  config, group = "large_data", clusters = c(TRUE, FALSE))
# You can even use clusters given to you for free in the `graph$nodes`
# data frame of `drake_graph_info()`.
vis_drake_graph(config, group = "status", clusters = "imported")
}
}
})

## End(Not run)
```

Index