Package 'drake'

January 4, 2021

Title A Pipeline Toolkit for Reproducible Computation at Scale

Version 7.13.0

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://docs.ropensci.org/drake/> and the online manual <https://books.ropensci.org/drake/>.

License GPL-3

URL https://github.com/ropensci/drake, https://docs.ropensci.org/drake/, https://books.ropensci.org/drake/

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

Depends R (>= 3.3.0)

Imports base64url, digest (>= 0.6.21), igraph, methods, parallel, rlang (>= 0.2.0), storr (>= 1.1.0), tidyselect (>= 1.0.0), txtq (>= 0.2.3), utils, vctrs (>= 0.2.0)

Suggests abind, bindr, callr, cli (>= 1.1.0), clustermq (>= 0.8.8), crayon, curl (>= 2.7), data.table, datasets, disk.frame, downloader, fst, future (>= 1.3.0), ggplot2, ggraph, grDevices, keras, knitr, lubridate, networkD3, prettycode, progress (>= 1.2.2), qs (>= 0.20.2), Rcpp, rmarkdown, rstudioapi, stats, styler (>= 1.2.0), testthat (>= 2.1.0), tibble, txtplot, usethis, visNetwork (>= 2.0.9), webshot

Encoding UTF-8

Language en-US

RoxygenNote 7.1.1.9000

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 Bond-Lamberty [ctb] (<https://orcid.org/0000-0001-9525-4633>),
Tristan Mahr [ctb] (<https://orcid.org/0000-0002-8890-5116>),
Miles McBain [ctb] (<https://orcid.org/0000-0003-2865-2548>),
Noam Ross [ctb] (<https://orcid.org/0000-0002-2136-0000>),
Ellis Hughes [ctb],
Matthew Mark Strasiotto [ctb],
Ben Marwick [rev],
Peter Slaughter [rev],
Eli Lilly and Company [cph]

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

Repository CRAN

Date/Publication 2021-01-04 16:20:02 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:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
library(drake)
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan) # Build everything.
plot(my_plan) # fast call to vis_drake_graph()
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)
}
```

```
# Dynamic branching
# Get the mean mpg for each cyl in the mtcars dataset.
plan <- drake_plan(
  raw = mtcars,
  group_index = raw$cyl,
  munged = target(raw[, c("mpg", "cyl")], dynamic = map(raw)),
  mean_mpg_by_cyl = target(
    data.frame(mpg = mean(munged$mpg), cyl = munged$cyl[1]),
    dynamic = group(munged, .by = group_index)
  )
)
make(plan)
readd(mean_mpg_by_cyl)
})

## End(Not run)
```

`bind_plans` *Row-bind together drake plans* **Stable**

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
```

`build_times` *See the time it took to build each target.* **Stable**

Description

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

Usage

```
build_times(
  ...,
  path = NULL,
  search = NULL,
  digits = 3,
  cache = drake::drake_cache(path = path),
  targets_only = NULL,
  verbose = NULL,
  jobs = 1,
  type = c("build", "command"),
  list = character(0)
)
```

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`	Path to a `drake` cache (usually a hidden .drake/ folder) or `NULL`.
`search`	Deprecated.
`digits`	How many digits to round the times to.
`cache`	drake cache. See `new_cache()`. If supplied, `path` is ignored.
`targets_only`	Deprecated.
`verbose`	Deprecated on 2019-09-11.
`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.
`list`	Character vector of targets to select.

Details

Times for dynamic targets (https://books.ropensci.org/drake/dynamic.html) only reflect the time it takes to post-process the sub-targets (typically very fast) and exclude the time it takes to build the sub-targets themselves. Sub-targets build times are listed individually.

Value

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

See Also

predict_runtime()

Examples

```
## Not run:
isolate_example("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.* **Stable**

Description

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

Usage

```
cached(
  ...,
  list = character(0),
  no_imported_objects = FALSE,
  path = NULL,
  search = NULL,
  cache = drake::drake_cache(path = path),
  verbose = NULL,
  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	Path to a drake cache (usually a hidden .drake/ folder) or NULL.
search	Deprecated.
cache	drake cache. See new_cache(). If supplied, path is ignored.
verbose	Deprecated on 2019-09-11.
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

cached_planned(), cached_unplanned(), readd(), loadd(), drake_plan(), make()

Examples

```
## Not run:
isolate_example("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)
```

cached_planned *List targets in both the plan and the cache.* **Stable**

Description

Includes dynamic sub-targets as well. See examples for details.

Usage

```
cached_planned(
  plan,
  path = NULL,
  cache = drake::drake_cache(path = path),
  namespace = NULL,
  jobs = 1
)
```

Arguments

plan	A drake plan.
path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.
cache	drake cache. See new_cache(). If supplied, path is ignored.
namespace	Character scalar, name of the storr namespace to use for listing objects.
jobs	Number of jobs/workers for parallel processing.

Value

A character vector of target and sub-target names.

See Also

cached(), cached_unplanned

Examples

```
## Not run:
isolate_example("cache_planned() example", {
plan <- drake_plan(w = 1)
make(plan)
cached_planned(plan)
plan <- drake_plan(
  x = seq_len(2),
  y = target(x, dynamic = map(x))
)
cached_planned(plan)
make(plan)
cached_planned(plan)
```

```
cached()
})

## End(Not run)
```

---

cached_unplanned *List targets in the cache but not the plan.* **Stable**

Description

Includes dynamic sub-targets as well. See examples for details.

Usage

```
cached_unplanned(
  plan,
  path = NULL,
  cache = drake::drake_cache(path = path),
  namespace = NULL,
  jobs = 1
)
```

Arguments

plan	A drake plan.
path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.
cache	drake cache. See new_cache(). If supplied, path is ignored.
namespace	Character scalar, name of the storr namespace to use for listing objects.
jobs	Number of jobs/workers for parallel processing.

Value

A character vector of target and sub-target names.

See Also

cached(), cached_planned

Examples

```
## Not run:
isolate_example("cache_unplanned() example", {
plan <- drake_plan(w = 1)
make(plan)
cached_unplanned(plan)
plan <- drake_plan(
  x = seq_len(2),
```

```
  y = target(x, dynamic = map(x))
)
cached_unplanned(plan)
make(plan)
cached_unplanned(plan)
# cached_unplanned() helps clean superfluous targets.
cached()
clean(list = cached_unplanned(plan))
cached()
})

## End(Not run)
```

`cancel` *Cancel a target mid-build* **Stable**

Description

Cancel a target mid-build. Upon cancellation, `drake` halts the current target and moves to the next one. The target's previous value and metadata, if they exist, remain in the cache.

Usage

```
cancel(allow_missing = TRUE)
```

Arguments

`allow_missing`	Logical. If `FALSE`, `drake` will not cancel the target if it is missing from the cache (or if you removed the key with `clean()`).

Value

Nothing.

See Also

cancel_if

Examples

```
## Not run:
isolate_example("cancel()", {
f <- function(x) {
  cancel()
  Sys.sleep(2) # Does not run.
}
g <- function(x) f(x)
plan <- drake_plan(y = g(1))
make(plan)
# Does not exist.
```

```
# readd(y)
})

## End(Not run)
```

`cancel_if` *Cancel a target mid-build under some condition* **Stable**)‘

Description

Cancel a target mid-build if some logical condition is met. Upon cancellation, `drake` halts the current target and moves to the next one. The target's previous value and metadata, if they exist, remain in the cache.

Usage

```
cancel_if(condition, allow_missing = TRUE)
```

Arguments

`condition`	Logical, whether to cancel the target.
`allow_missing`	Logical. If `FALSE`, `drake` will not cancel the target if it is missing from the cache (or if you removed the key with `clean()`).

Value

Nothing.

See Also

cancel

Examples

```
## Not run:
isolate_example("cancel_if()", {
f <- function(x) {
  cancel_if(x > 1)
  Sys.sleep(2) # Does not run if x > 1.
}
g <- function(x) f(x)
plan <- drake_plan(y = g(2))
make(plan)
# Does not exist.
# readd(y)
})

## End(Not run)
```

clean *Invalidate and deregister targets.* **Stable**

Description

Force targets to be out of date and remove target names from the data in the cache. Be careful and run `which_clean()` before `clean()`. That way, you know beforehand which targets will be compromised.

Usage

```
clean(
  ...,
  list = character(0),
  destroy = FALSE,
  path = NULL,
  search = NULL,
  cache = drake::drake_cache(path = path),
  verbose = NULL,
  jobs = NULL,
  force = FALSE,
  garbage_collection = FALSE,
  purge = FALSE
)
```

Arguments

`...`	Symbols, individual targets to remove.
`list`	Character vector of individual targets to 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`	Path to a `drake` cache (usually a hidden .drake/ folder) or `NULL`.
`search`	Deprecated
`cache`	drake cache. See `new_cache()`. If supplied, `path` is ignored.
`verbose`	Deprecated
`jobs`	Deprecated.
`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() invalidates **all** targets, so be careful. clean() always:

1. Forces targets to be out of date so the next make() does not skip them.
2. Deregisters targets so loadd(your_target) and readd(your_target) no longer work.

By default, clean() does not actually remove the underlying data. Even old targets from the distant past are still in the cache and recoverable via drake_history() and make(recover = TRUE). To actually remove target data from the cache, as well as any file_out() files from any targets you are currently cleaning, run clean(garbage_collection = TRUE). Garbage collection is slow, but it reduces the storage burden of the cache.

Value

Invisibly return NULL.

See Also

which_clean(), drake_gc()

Examples

```
## Not run:
isolate_example("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.
# Show all registered targets in the cache.
cached()
# Deregister 'summ_regression1_large' and 'small' in the cache.
clean(summ_regression1_large, small)
# Those objects are no longer registered as targets.
cached()
# Rebuild the invalidated/outdated targets.
make(my_plan)
# Clean everything.
clean()
# But the data objects and files are not actually gone!
file.exists("report.md")
drake_history()
make(my_plan, recover = TRUE)
# You need garbage collection to actually remove the data
# and any file_out() files of any uncleaned targets.
clean(garbage_collection = TRUE)
drake_history()
make(my_plan, recover = TRUE)
}
})

## End(Not run)
```

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

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:
isolate_example("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.
outdated(my_plan) # Which targets are out of date?
# Run the workflow to build all the targets in the plan.
make(my_plan)
outdated(my_plan) # 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_function *Turn a script into a function.* **Stable**

Description

code_to_function() is a quick (and very dirty) way to retrofit drake to an existing script-based project. It parses individual *.R/*.RMD files into functions so they can be added into the drake workflow.

Usage

```
code_to_function(path, envir = parent.frame())
```

Arguments

path	Character vector, path to script.
envir	Environment of the created function.

Details

Most data science workflows consist of imperative scripts. drake, on the other hand, assumes you write *functions*. code_to_function() allows for pre-existing workflows to incorporate drake as a workflow management tool seamlessly for cases where re-factoring is unfeasible. So drake can monitor dependencies, the targets are passed as arguments of the dependent functions.

Value

A function to be input into the drake plan

See Also

file_in(), file_out(), knitr_in(), ignore(), no_deps(), code_to_plan(), plan_to_code(), plan_to_notebook()

Examples

```
## Not run:
isolate_example("contain side effects", {
if (requireNamespace("ggplot2", quietly = TRUE)) {
# The `code_to_function()` function creates a function that makes it
# available for drake to process as part of the workflow.
# The main purpose is to allow pre-existing workflows to incorporate drake
# into the workflow seamlessly for cases where re-factoring is unfeasible.
#

script1 <- tempfile()
script2 <- tempfile()
script3 <- tempfile()
```

```
script4 <- tempfile()

writeLines(c(
  "data <- mtcars",
  "data$make <- do.call('c',",
  "lapply(strsplit(rownames(data), split=\" \"), `[`, 1))",
  "saveRDS(data, \"mtcars_alt.RDS\")"
 ),
  script1
)

writeLines(c(
  "data <- readRDS(\"mtcars_alt.RDS\")",
  "mtcars_lm <- lm(mpg~cyl+disp+vs+gear+make,data=data)",
  "saveRDS(mtcars_lm, \"mtcars_lm.RDS\")"
  ),
  script2
)
writeLines(c(
  "mtcars_lm <- readRDS(\"mtcars_lm.RDS\")",
  "lm_summary <- summary(mtcars_lm)",
  "saveRDS(lm_summary, \"mtcars_lm_summary.RDS\")"
  ),
  script3
)
writeLines(c(
  "data<-readRDS(\"mtcars_alt.RDS\")",
  "gg <- ggplot2::ggplot(data)+",
  "ggplot2::geom_point(ggplot2::aes(",
  "x=disp, y=mpg, shape=as.factor(vs), color=make))",
  "ggplot2::ggsave(\"mtcars_plot.png\", gg)"
 ),
  script4
)


do_munge <- code_to_function(script1)
do_analysis <- code_to_function(script2)
do_summarize <- code_to_function(script3)
do_vis <- code_to_function(script4)

plan <- drake_plan(
  munged   = do_munge(),
  analysis = do_analysis(munged),
  summary  = do_summarize(analysis),
  plot     = do_vis(munged)
 )

plan
# drake knows  "script1" is the first script to be evaluated and ran,
# because it has no dependencies on other code and a dependency of
# `analysis`. See for yourself:
```

```
make(plan)

# See the connections that the sourced scripts create:
if (requireNamespace("visNetwork", quietly = TRUE)) {
  vis_drake_graph(plan)
}
}
})

## End(Not run)
```

`code_to_plan` *Turn an R script file or* `knitr` */ R Markdown report into a* `drake` *plan.*
Questioning

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(Ozone ~ Temp + Wind, 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.
code_to_plan(file)
```

---

deps_code *List the dependencies of a function or command* **Stable**

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.* **Stable**

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:
isolate_example("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.* **Stable**

Description

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

- the command changed (deps_profile() reports the *hash* of the command, not the command itself)
- 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).
- the pseudo-random number generator seed changed. Unfortunately, deps_profile() does not currently get more specific than that.

Usage

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

Arguments

target	Name of the target.
...	Arguments to make(), such as plan and targets.
character_only	Logical, whether to assume target is a character string rather than a symbol.
config	Deprecated.

Value

A data frame of old and new values for each of the main triggers, along with an indication of which values changed since the last make().

See Also

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

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Load drake's canonical example.
make(my_plan) # Run the project, build the targets.
# Get some example dependency profiles of targets.
deps_profile(small, my_plan)
# 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, my_plan)
}
})

## End(Not run)
```

`deps_target` *List the dependencies of a target* **Stable**

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, ..., character_only = FALSE, config = NULL)
```

Arguments

`target`	A symbol denoting a target name, or if `character_only` is TRUE, a character scalar denoting a target name.
`...`	Arguments to `make()`, such as `plan` and `targets`.
`character_only`	Logical, whether to assume target is a character string rather than a symbol.
`config`	Deprecated.

Value

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

See Also

`deps_code()`, `deps_knitr()`

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
deps_target(regression1_small, my_plan)
})

## End(Not run)
```

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

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 = NULL,
  search = NULL,
  cache = drake::drake_cache(path = path),
  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`	Path to a `drake` cache (usually a hidden .drake/ folder) or `NULL`.
`search`	Deprecated.
`cache`	drake cache. See `new_cache()`. If supplied, `path` is ignored.
`verbose`	Deprecated on 2019-09-11.

Value

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

See Also

`drake_failed()`, `drake_progress()`, `readd()`, `drake_plan()`, `make()`

Examples

```
## Not run:
isolate_example("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)
drake_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 rlang::trace_back() function).
})

## End(Not run)
```

---

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

Description

Not valid for dynamic branching.

Usage

```
drake_build(
  target,
  ...,
  meta = NULL,
  character_only = FALSE,
  replace = FALSE,
  config = NULL
)
```

Arguments

target	Name of the target.
...	Arguments to make(), such as the plan and environment.

meta	Deprecated.
character_only	Logical, whether name should be treated as a character or a symbol (just like character.only in library()).
replace	Logical. If FALSE, items already in your environment will not be replaced.
config	Deprecated 2019-12-22.

Value

The value of the target right after it is built.

See Also

drake_debug()

Examples

```
## Not run:
isolate_example("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").
out <- drake_build(small, my_plan)
# Now includes `small`.
cached()
head(readd(small))
# `small` was invisibly returned.
head(out)
}
})

## End(Not run)
```

---

drake_cache *Get the cache of a* drake *project.* **Stable**

Description

make() saves the values of your targets so you rarely need to think about output files. By default, the cache is a hidden folder called .drake/. You can also supply your own storr cache to the cache argument of make(). The drake_cache() function retrieves this cache.

Usage

```
drake_cache(path = NULL, verbose = NULL, console_log_file = NULL)
```

Arguments

path	Character. Set path to the path of a storr::storr_rds() cache to retrieve a specific cache generated by storr::storr_rds() or drake::new_cache(). If the path argument is NULL, drake_cache() searches up through parent directories to find a folder called .drake/.
verbose	Deprecated on 2019-09-11.
console_log_file	Deprecated on 2019-09-11.

Details

drake_cache() actually returns a *decorated* storr, an object that *contains* a storr (plus bells and whistles). To get the *actual* inner storr, use drake_cache()$storr. Most methods are delegated to the inner storr. Some methods and objects are new or overwritten. Here are the ones relevant to users.

- history: drake's history (which powers drake_history()) is a txtq. Access it with drake_cache()$history.
- import(): The import() method is a function that can import targets, function dependencies, etc. from one decorated storr to another. History is not imported. For that, you have to work with the history txtqs themselves, Arguments to import():
  - ... and list: specify targets to import just like with loadd(). Leave these blank to import everything.
  - from: the decorated storr from which to import targets.
  - jobs: number of local processes for parallel computing.
  - gc: TRUE or FALSE, whether to run garbage collection for memory after importing each target. Recommended, but slow.
- export(): Same as import(), except the from argument is replaced by to: the decorated storr where the targets end up.

Value

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

See Also

new_cache(), drake_config()

Examples

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

```
y <- storr::storr_rds(".drake") # Nearly equivalent.
# List the objects readable from the cache with readd().
x$list()
# drake_cache() actually returns a *decorated* storr.
# The *real* storr is inside.
drake_cache()$storr
}
# You can import and export targets to and from decorated storrs.
plan1 <- drake_plan(w = "w", x = "x")
plan2 <- drake_plan(a = "a", x = "x2")
cache1 <- new_cache("cache1")
cache2 <- new_cache("cache2")
make(plan1, cache = cache1)
make(plan2, cache = cache2)
cache1$import(cache2, a)
cache1$get("a")
cache1$get("x")
cache1$import(cache2)
cache1$get("x")
# With txtq >= 0.1.6.9002, you can import history from one cache into
# another.
# nolint start
# drake_history(cache = cache1)
# cache1$history$import(cache2$history)
# drake_history(cache = cache1)
# nolint end
})

## End(Not run)
```

`drake_cache_log` *Get the state of the cache.* **Stable**

Description

Get the fingerprints of all the targets in a data frame. 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 = NULL,
  search = NULL,
  cache = drake::drake_cache(path = path),
  verbose = 1L,
  jobs = 1,
  targets_only = FALSE
)
```

Arguments

path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.
search	Deprecated.
cache	drake cache. See new_cache(). If supplied, path is ignored.
verbose	Deprecated on 2019-09-11.
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(), drake_cache()

Examples

```
## Not run:
isolate_example("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_cancelled *List cancelled targets.* **Stable**

Description

List the targets that were cancelled in the current or previous call to `make()` using `cancel()` or `cancel_if()`.

Usage

```
drake_cancelled(cache = drake::drake_cache(path = path), path = NULL)
```

Arguments

cache	drake cache. See `new_cache()`. If supplied, `path` is ignored.
path	Path to a `drake` cache (usually a hidden .drake/ folder) or `NULL`.

Value

A character vector of target names.

See Also

`drake_running()`, `drake_failed()`, `make()`

Examples

```
## Not run:
isolate_example("contain side effects", {
plan <- drake_plan(x = 1, y = cancel_if(x > 0))
make(plan)
drake_cancelled()
```

```
})

## End(Not run)
```

---

drake_config *Ending of _drake.R for r_make() and friends* **Stable**

Description

Call this function inside the _drake.R script for `r_make()` and friends. All non-deprecated function arguments are the same between `make()` and `drake_config()`.

Usage

```
drake_config(
  plan,
  targets = NULL,
  envir = parent.frame(),
  verbose = 1L,
  hook = NULL,
  cache = drake::drake_cache(),
  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 = TRUE,
  graph = NULL,
  trigger = drake::trigger(),
  skip_targets = FALSE,
  skip_imports = FALSE,
  skip_safety_checks = FALSE,
  lazy_load = "eager",
  session_info = NULL,
  cache_log_file = NULL,
  seed = NULL,
```

```
  caching = c("main", "master", "worker"),
  keep_going = FALSE,
  session = NULL,
  pruning_strategy = NULL,
  makefile_path = NULL,
  console_log_file = NULL,
  ensure_workers = NULL,
  garbage_collection = FALSE,
  template = list(),
  sleep = function(i) 0.01,
  hasty_build = NULL,
  memory_strategy = "speed",
  spec = NULL,
  layout = NULL,
  lock_envir = TRUE,
  history = TRUE,
  recover = FALSE,
  recoverable = TRUE,
  curl_handles = list(),
  max_expand = NULL,
  log_build_times = TRUE,
  format = NULL,
  lock_cache = TRUE,
  log_make = NULL,
  log_worker = FALSE
)
```

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. You may supply static and/or whole dynamic targets, but no sub-targets.
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	Integer, control printing to the console/terminal. • `0`: print nothing. • `1`: print target-by-target messages as `make()` progresses.

- 2: show a progress bar to track how many targets are done so far.

hook — Deprecated.

cache — drake cache as created by new_cache(). See also drake_cache().

fetch_cache — Deprecated.

parallelism — Character scalar, type of parallelism to use. For detailed explanations, see the high-performance computing chapter # nolint 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() 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://books.ropensci.org/drake/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 extra files in the cache (usually the .drake/ folder) and slows down make() a little. If you need to reduce or limit the number of files in the cache, call make(log_progress = FALSE, recover = FALSE).

graph
: Deprecated.

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
: An old feature, currently being questioned. For the current recommendations on memory management, see https://books.ropensci.org/drake/memory.html#memory-strategies. The lazy_load argument is either a character vector or a logical. For dynamic targets, the behavior is always "eager" (see below). So the lazy_load argument is for static targets only. Choices for lazy_load:

  - "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".

  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. Defaults to TRUE. 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 CSV cache log file to write. If TRUE, the default file name is used (drake_cache.CSV). 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 exception to this last point is make(parallelism = "clustermq") 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, either "main" or "worker".

- "main": Targets are built by remote workers and sent back to the main process. Then, the main 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 Deprecated.

console_log_file
Deprecated in favor of log_make.

`ensure_workers` Deprecated.

`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")`), 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` Optional function on a single numeric argument `i`. Default: `function(i) 0.01`.

To conserve memory, `drake` assigns a brand new closure to `sleep`, so your custom function should not depend on in-memory data except from loaded packages.

For parallel processing, `drake` uses a central main 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 main process sleeps to avoid throttling. The `sleep` argument to `make()` and `drake_config()` allows you to customize how much time the main 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` Deprecated

`memory_strategy`

Character scalar, name of the strategy `drake` uses to load/unload a target's dependencies in memory. You can give each target its own memory strategy, (e.g. `drake_plan(x = 1,y = target(f(x),memory_strategy = "lookahead"))`) to override the global memory strategy. Choices:

- `"speed"`: Once a target is newly built or loaded in memory, just keep it there. This choice maximizes speed and hogs memory.
- `"autoclean"`: Just before building each new target, unload everything from memory except the target's direct dependencies. After a target is built, discard it from memory. (Set `garbage_collection = TRUE` to make sure it is really gone.) This option conserves memory, but it sacrifices speed because each new target needs to reload any previously unloaded targets from storage.
- `"preclean"`: Just before building each new target, unload everything from memory except the target's direct dependencies. After a target is built, keep it in memory until `drake` determines they can be unloaded. 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. After a target is built, keep it in memory until the next memory management stage. 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.
- "unload": Just before building each new target, unload all targets from memory. After a target is built, **do not** keep it in memory. This mode aggressively optimizes for both memory and speed, but in commands and triggers, you have to manually load any dependencies you need using readd().
- "none": Do not manage memory at all. Do not load or unload anything before building targets. After a target is built, **do not** keep it in memory. This mode aggressively optimizes for both memory and speed, but in commands and triggers, you have to manually load any dependencies you need using readd().

For even more direct control over which targets drake keeps in memory, see the help file examples of drake_envir(). Also see the garbage_collection argument of make() and drake_config().

spec — Deprecated.

layout — Deprecated.

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.

history — Logical, whether to record the build history of your targets. You can also supply a txtq, which is how drake records history. Must be TRUE for drake_history() to work later.

recover — Logical, whether to activate automated data recovery. The default is FALSE because

1. Automated data recovery is still stable.
2. It has reproducibility issues. Targets recovered from the distant past may have been generated with earlier versions of R and earlier package environments that no longer exist.
3. It is not always possible, especially when dynamic files are combined with dynamic branching (e.g. dynamic = map(stuff) and format = "file" etc.) since behavior is harder to predict in advance.

How it works: if recover is TRUE, drake tries to salvage old target values from the cache instead of running commands from the plan. A target is recoverable if

1. There is an old value somewhere in the cache that shares the command, dependencies, etc. of the target about to be built.
2. The old value was generated with make(recoverable = TRUE).

If both conditions are met, drake will

1. Assign the most recently-generated admissible data to the target, and
2. skip the target's command.

Functions `recoverable()` and `r_recoverable()` show the most upstream outdated targets that will be recovered in this way in the next `make()` or `r_make()`.

`recoverable` Logical, whether to make target values recoverable with `make(recover = TRUE)`. This requires writing extra files to the cache, and it prevents old metadata from being removed with garbage collection (`clean(garbage_collection = TRUE)`, `gc()` in `storrs`). If you need to limit the cache size or the number of files in the cache, consider `make(recoverable = FALSE, progress = FALSE)`. Recovery is not always possible, especially when dynamic files are combined with dynamic branching (e.g. `dynamic = map(stuff)` and `format = "file"` etc.) since behavior is harder to predict in advance.

`curl_handles` A named list of curl handles. Each value is an object from `curl::new_handle()`, and each name is a URL (and should start with "http", "https", or "ftp"). Example: list(http://httpbin.org/basic-auth = curl::new_handle(username = "user", password = "passwd")) Then, if your plan has `file_in("http://httpbin.org/basic-auth/user/pas` `drake` will authenticate using the username and password of the handle for http://httpbin.org/basic-auth/.

`drake` uses partial matching on text to find the right handle of the `file_in()` URL, so the name of the handle could be the complete URL (`"http://httpbin.org/basic-auth/user/` or a part of the URL (e.g. `"http://httpbin.org/"` or `"http://httpbin.org/basic-auth/"`). If you have multiple handles whose names match your URL, `drake` will choose the closest match.

`max_expand` Positive integer, optional. `max_expand` is the maximum number of targets to generate in each `map()`, `cross()`, or `group()` dynamic transform. Useful if you have a massive number of dynamic sub-targets and you want to work with only the first few sub-targets before scaling up. Note: the `max_expand` argument of `make()` and `drake_config()` is for dynamic branching only. The static branching `max_expand` is an argument of `drake_plan()` and `transform_plan()`.

`log_build_times` Logical, whether to record build_times for targets. Mac users may notice a 20% speedup in `make()` with `build_times = FALSE`.

`format` Character, an optional custom storage format for targets without an explicit `target(format = ...)` in the plan. Details about formats: `https://books.ropensci.org/drake/plans.html#special-data-formats-for-targets` # nolint

`lock_cache` Logical, whether to lock the cache before running `make()` etc. It is usually recommended to keep cache locking on. However, if you interrupt `make()` before it can clean itself up, then the cache will stay locked, and you will need to manually unlock it with `drake::drake_cache("xyz")$unlock()`. Repeatedly unlocking the cache by hand is annoying, and `lock_cache = FALSE` prevents the cache from locking in the first place.

`log_make` Optional character scalar of a file name or connection object (such as `stdout()`) to dump maximally verbose log information for `make()` and other functions (all functions that accept a `config` argument, plus `drake_config()`). If you choose

to use a text file as the console log, it will persist over multiple function calls until you delete it manually. Fields in each row the log file, from left to right: - The node name (short host name) of the computer (from `Sys.info()["nodename"]`). - The process ID (from `Sys.getpid()`). - A timestamp with the date and time (in microseconds). - A brief description of what `drake` was doing. The fields are separated by pipe symbols ("|"‘).

`log_worker` Logical, same as the `log_worker` argument of `clustermq::workers()` and `clustermq::Q()`. Only relevant if `parallelism` is `"clustermq"`.

Details

In `drake`, `make()` has two stages:

1. Configure a workflow to your environment and plan.
2. Build targets. The `drake_config()` function just does step (1), which is a common requirement for not only `make()`, but also utility functions like `vis_drake_graph()` and `outdated()`. That is why `drake_config()` is a requirement for the _drake.R script, which powers `r_make()`, `r_outdated()`, `r_vis_drake_graph()`, etc.

Value

A configured `drake` workflow.

Recovery

`make(recover = TRUE, recoverable = TRUE)` powers automated data recovery. The default of `recover` is `FALSE` because targets recovered from the distant past may have been generated with earlier versions of R and earlier package environments that no longer exist.

How it works: if `recover` is `TRUE`, `drake` tries to salvage old target values from the cache instead of running commands from the plan. A target is recoverable if

1. There is an old value somewhere in the cache that shares the command, dependencies, etc. of the target about to be built.
2. The old value was generated with `make(recoverable = TRUE)`.

If both conditions are met, `drake` will

1. Assign the most recently-generated admissible data to the target, and
2. skip the target's command.

See Also

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

Examples

```
## Not run:
isolate_example("quarantine side effects", {
if (requireNamespace("knitr", quietly = TRUE)) {
writeLines(
  c(
    "library(drake)",
    "load_mtcars_example()",
    "drake_config(my_plan, targets = c(\"small\", \"large\"))"
  ),
  "_drake.R" # default value of the `source` argument
)
cat(readLines("_drake.R"), sep = "\n")
r_outdated()
r_make()
r_outdated()
}
})

## End(Not run)
```

---

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

Description

Not valid for dynamic branching. 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,
  ...,
  character_only = FALSE,
  replace = FALSE,
  verbose = TRUE,
  config = NULL
)
```

Arguments

target	Name of the target.
...	Arguments to make(), such as the plan and environment.
character_only	Logical, whether name should be treated as a character or a symbol (just like character.only in library()).

replace	Logical. If FALSE, items already in your environment will not be replaced.
verbose	Logical, whether to print out the target you are debugging.
config	Deprecated 2019-12-22.

Value

The value of the target right after it is built.

See Also

drake_build()

Examples

```
## Not run:
isolate_example("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").
# out <- drake_debug(small, my_plan)
# `small` was invisibly returned.
# head(out)
}
})

## End(Not run)
```

---

drake_done *List done targets.* **Stable**

Description

List the targets that completed in the current or previous call to make().

Usage

```
drake_done(cache = drake::drake_cache(path = path), path = NULL)
```

Arguments

cache	drake cache. See new_cache(). If supplied, path is ignored.
path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.

Value

A character vector of target names.

See Also

drake_running(), drake_failed(), drake_cancelled(), drake_progress(), make()

Examples

```
## Not run:
isolate_example("contain side effects", {
plan <- drake_plan(x = 1, y = x)
make(plan)
drake_done()
})

## End(Not run)
```

---

drake_envir *Get the environment where drake builds targets* **Questioning**

Description

Call this function inside the commands in your plan to get the environment where drake builds targets. Advanced users can use it to strategically remove targets from memory while make() is running.

Usage

```
drake_envir(which = c("targets", "dynamic", "subtargets", "imports"))
```

Arguments

which	Character of length 1, which environment to select. See the details of this help file.

Details

drake manages in-memory targets in 4 environments: one with sub-targets, one with whole dynamic targets, one with static targets, and one with imported global objects and functions. This last environment is usually the environment from which you call make(). Select the appropriate environment for your use case with the which argument of drake_envir().

Value

The environment where drake builds targets.

Keywords

drake_plan() understands special keyword functions for your commands. With the exception of target(), each one is a proper function with its own help file.

- target(): give the target more than just a command. Using target(), you can apply a transformation (examples: https://books.ropensci.org/drake/plans.html#large-plans), # nolint supply a trigger (https://books.ropensci.org/drake/triggers.html), # nolint or set any number of custom columns.
- file_in(): declare an input file dependency.
- file_out(): declare an output file to be produced when the target is built.
- knitr_in(): declare a knitr file dependency such as an R Markdown (*.Rmd) or R LaTeX (*.Rnw) file.
- ignore(): force drake to entirely ignore a piece of code: do not track it for changes and do not analyze it for dependencies.
- no_deps(): tell drake to not track the dependencies of a piece of code. drake still tracks the code itself for changes.
- id_chr(): Get the name of the current target.
- drake_envir(): get the environment where drake builds targets. Intended for advanced custom memory management.

See Also

from_plan()

Examples

```
## Not run:
isolate_example("contain side effects", {
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 = parent.env(drake_envir())))
    # We don't need the large_data_* targets in memory anymore.
    rm(large_data_1, large_data_2, envir = drake_envir("targets"))
    print(ls(envir = drake_envir("targets")))
    mean(subset)
  }
)
make(plan, cache = storr::storr_environment(), session_info = FALSE)
})

## End(Not run)
```

drake_example *Download the files of an example* drake *project.* **Stable**

Description

The drake_example() function downloads a folder from https://github.com/wlandau/drake-examples. By default, it creates a new folder with the example name in your current working directory. After the files are written, have a look at the enclosed README file. Other instructions are available in the files at https://github.com/wlandau/drake-examples.

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:
isolate_example("Quarantine side effects.", {
if (requireNamespace("downloader")) {
drake_examples() # List all the drake examples.
# Sets up the same example from load_mtcars_example()
```

```
drake_example("mtcars")
# Sets up the SLURM example.
drake_example("slurm")
}
})

## End(Not run)
```

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

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:
isolate_example("Quarantine side effects.", {
if (requireNamespace("downloader")) {
drake_examples() # List all the drake examples.
# Sets up the example from load_mtcars_example()
drake_example("mtcars")
# Sets up the SLURM example.
drake_example("slurm")
}
})

## End(Not run)
```

drake_failed *List failed targets.* **Stable**

Description

List the targets that quit in error during `make()`.

Usage

```
drake_failed(cache = drake::drake_cache(path = path), path = NULL)
```

Arguments

cache	drake cache. See `new_cache()`. If supplied, `path` is ignored.
path	Path to a `drake` cache (usually a hidden .drake/ folder) or `NULL`.

Value

A character vector of target names.

See Also

`drake_done()`, `drake_running()`, `drake_cancelled()`, `drake_progress()`, `make()`

Examples

```
## Not run:
isolate_example("contain side effects", {
if (suppressWarnings(require("knitr"))) {
# Build a plan doomed to fail:
bad_plan <- drake_plan(x = function_doesnt_exist())
cache <- storr::storr_environment() # optional
try(
  make(bad_plan, cache = cache, history = FALSE),
  silent = TRUE
) # error
drake_failed(cache = cache) # "x"
e <- diagnose(x, cache = cache) # Retrieve the cached error log of x.
names(e)
e$error
names(e$error)
}
})

## End(Not run)
```

drake_gc *Do garbage collection on the drake cache.* **Stable**

Description

Garbage collection removes obsolete target values from the cache.

Usage

```
drake_gc(
  path = NULL,
  search = NULL,
  verbose = NULL,
  cache = drake::drake_cache(path = path),
  force = FALSE
)
```

Arguments

path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.
search	Deprecated.
verbose	Deprecated on 2019-09-11.
cache	drake cache. See new_cache(). If supplied, path is ignored.
force	Logical, whether to load the cache despite any back compatibility issues with the running version of drake.

Details

Caution: garbage collection *actually* removes data so it is no longer recoverable with drake_history() or make(recover = TRUE). You cannot undo this operation. Use at your own risk.

Value

NULL

See Also

clean()

Examples

```
## Not run:
isolate_example("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` *Session info of the last call to* `make()`. **Stable**

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 = NULL,
  search = NULL,
  cache = drake::drake_cache(path = path),
  verbose = 1L
)
```

Arguments

`path`	Path to a `drake` cache (usually a hidden .drake/ folder) or `NULL`.
`search`	Deprecated.
`cache`	drake cache. See `new_cache()`. If supplied, `path` is ignored.
`verbose`	Deprecated on 2019-09-11.

Value

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

See Also

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

Examples

```
## Not run:
isolate_example("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 *Visualize the workflow with* ggraph/ggplot2 **Stable**

Description

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

Usage

```
drake_ggraph(
  ...,
  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,
  label_nodes = FALSE,
  transparency = TRUE,
  config = NULL
)
```

Arguments

...	Arguments to make(), such as plan and targets.

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 original drake 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.
label_nodes	Logical, whether to label the nodes. If FALSE, the graph will not have any text next to the nodes, which is recommended for large graphs with lots of targets.
transparency	Logical, whether to allow transparency in the rendered graph. Set to FALSE if you get warnings like "semi-transparency is not supported on this device".
config	Deprecated.

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(), text_drake_graph()

Examples

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

## End(Not run)
```

---

drake_graph_info *Prepare the workflow graph for visualization* **Stable**

Description

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

Usage

```
drake_graph_info(
  ...,
  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,
  on_select_col = NULL,
```

```
  config = NULL
)
```

Arguments

...	Arguments to make(), such as plan and targets.
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 original drake 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.

on_select_col
: Optional string corresponding to the column name in the plan that should provide data for the on_select event.

config
: Deprecated.

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:
isolate_example("Quarantine side effects.", {
if (requireNamespace("visNetwork", quietly = TRUE)) {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
vis_drake_graph(my_plan)
# 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(my_plan)
# Choose a subset of the graph.
smaller_raw_graph <- drake_graph_info(
  my_plan,
  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')
}
}
})

## End(Not run)
```

drake_history *History and provenance* **Stable**

Description

See the history and provenance of your targets: what you ran, when you ran it, the function arguments you used, and how to get old data back.

Usage

```
drake_history(cache = NULL, history = NULL, analyze = TRUE, verbose = NULL)
```

Arguments

cache	drake cache as created by new_cache(). See also drake_cache().
history	Logical, whether to record the build history of your targets. You can also supply a txtq, which is how drake records history. Must be TRUE for drake_history() to work later.
analyze	Logical, whether to analyze drake_plan() commands for arguments to function calls. Could be slow because this requires parsing and analyzing lots of R code.
verbose	Deprecated on 2019-09-11.

Details

drake_history() returns a data frame with the following columns.

- target: the name of the target.
- current: logical, whether the row describes the data actually assigned to the target name in the cache, e.g. what you get with loadd(target) and readd(target). Does **NOT** tell you if the target is up to date.
- built: when the target's value was stored in the cache. This is the true creation date of the target's value, not the recovery date from make(recover = TRUE).
- exists: logical, whether the target's historical value still exists in the cache. Garbage collection via (clean(garbage_collection = TRUE) and drake_cache()$gc()) remove these historical values, but clean() under the default settings does not.
- hash: fingerprint of the target's historical value in the cache. If the value still exists, you can read it with drake_cache()$get_value(hash).
- command: the drake_plan() command executed to build the target.
- seed: random number generator seed.
- runtime: the time it took to execute the drake_plan() command. Does not include overhead due to drake's processing.

If analyze is TRUE, various other columns are included to show the explicitly-named length-1 arguments to function calls in the commands. See the "Provenance" section for more details.

Value

A data frame of target history.

Provenance

If `analyze` is `TRUE`, `drake` scans your `drake_plan()` commands for function arguments and mentions them in the history. A function argument shows up if and only if: 1. It has length 1.
2. It is atomic, i.e. a base type: logical, integer, real, complex, character, or raw.
3. It is explicitly named in the function call, For example, `x` is detected as `1` in `fn(list(x = 1))` but not `f(list(1))`. The exceptions are `file_out()`, `file_in()`, and `knitr_in()`. For example, `filename` is detected as `"my_file.csv"` in `process_data(filename = file_in("my_file.csv"))`. NB: in `process_data(filename = file_in("a","b"))` `filename` is not detected because the value must be atomic.

Examples

```
## Not run:
isolate_example("contain side-effects", {
if (requireNamespace("knitr", quietly = TRUE)) {
# First, let's iterate on a drake workflow.
load_mtcars_example()
make(my_plan, history = TRUE, verbose = 0L)
# Naturally, we'll make updates to our targets along the way.
reg2 <- function(d) {
  d$x2 <- d$x ^ 3
  lm(y ~ x2, data = d)
}
Sys.sleep(0.01)
make(my_plan, history = TRUE, verbose = 0L)
# The history is a data frame about all the recorded runs of your targets.
out <- drake_history(analyze = TRUE)
print(out)
# Let's use the history to recover the oldest version
# of our regression2_small target.
oldest_reg2_small <- max(which(out$target == "regression2_small"))
hash_oldest_reg2_small <- out[oldest_reg2_small, ]$hash
cache <- drake_cache()
cache$get_value(hash_oldest_reg2_small)
# If you run clean(), drake can still find all the targets.
clean(small)
drake_history()
# But if you run clean() with garbage collection,
# older versions of your targets may be gone.
clean(large, garbage_collection = TRUE)
drake_history()
invisible()
}
})

## End(Not run)
```

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

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:
plan <- drake_plan(x = rnorm(1e7), y = rnorm(1e7))
# List the available template files.
drake_hpc_template_files()
# Write a SLURM template file.
out <- file.path(tempdir(), "slurm_batchtools.tmpl")
drake_hpc_template_file("slurm_batchtools.tmpl", to = tempdir())
cat(readLines(out), sep = "\n")
# library(future.batchtools) # nolint
# future::plan(batchtools_slurm, template = out) # nolint
# make(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. **Stable**

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:
plan <- drake_plan(x = rnorm(1e7), y = rnorm(1e7))
# List the available template files.
drake_hpc_template_files()
# Write a SLURM template file.
out <- file.path(tempdir(), "slurm_batchtools.tmpl")
drake_hpc_template_file("slurm_batchtools.tmpl", to = tempdir())
cat(readLines(out), sep = "\n")
# library(future.batchtools) # nolint
# future::plan(batchtools_slurm, template = out) # nolint
# make(plan, parallelism = "future", jobs = 2) # nolint

## End(Not run)
```

drake_plan

Create a drake plan for the `plan` *argument of* `make()`. **Stable**

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.

Usage

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

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`	Deprecated. Use `tidy_eval` instead.
`transform`	Logical, whether to transform the plan into a larger plan with more targets. Requires the `transform` field in `target()`. See the examples for details.
`trace`	Logical, whether to add columns to show what happens during target transformations.
`envir`	Environment for tidy evaluation.
`tidy_eval`	Logical, whether to use tidy evaluation (e.g. unquoting/!!) when resolving commands. Tidy evaluation in transformations is always turned on regardless of the value you supply to this argument.
`max_expand`	Positive integer, optional. `max_expand` is the maximum number of targets to generate in each `map()`, `split()`, or `cross()` transform. Useful if you have a massive plan and you want to test and visualize a strategic subset of targets before scaling up. Note: the `max_expand` argument of `drake_plan()` and `transform_plan()` is for static branching only. The dynamic branching `max_expand` is an argument of `make()` and `drake_config()`.

Details

Besides `"target"` and `"command"`, `drake_plan()` understands a special set of optional columns. For details, visit https://books.ropensci.org/drake/plans.html#special-custom-columns-in-your-plan
nolint

Value

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

Columns

drake_plan() creates a special data frame. At minimum, that data frame must have columns target and command with the target names and the R code chunks to build them, respectively.

You can add custom columns yourself, either with target() (e.g. drake_plan(y = target(f(x), transform = map(c(1, 2)), format = "fst"))) or by appending columns post-hoc (e.g. plan$col <- vals).

Some of these custom columns are special. They are optional, but drake looks for them at various points in the workflow.

- transform: a call to map(), split(), cross(), or combine() to create and manipulate large collections of targets. Details: (https://books.ropensci.org/drake/plans.html#large-plans). # nolint
- format: set a storage format to save big targets more efficiently. See the "Formats" section of this help file for more details.
- trigger: rule to decide whether a target needs to run. It is recommended that you define this one with target(). Details: https://books.ropensci.org/drake/triggers.html.
- hpc: logical values (TRUE/FALSE/NA) whether to send each target to parallel workers. Visit https://books.ropensci.org/drake/hpc.html#selectivity to learn more.
- resources: target-specific lists of resources for a computing cluster. See https://books.ropensci.org/drake/hpc.html#advanced-options for details.
- caching: overrides the caching argument of make() for each target individually. Possible values:
  - "main": tell the main process to store the target in the cache.
  - "worker": tell the HPC worker to store the target in the cache.
  - NA: default to the caching argument of make().
- elapsed and cpu: number of seconds to wait for the target to build before timing out (elapsed for elapsed time and cpu for CPU time).
- retries: number of times to retry building a target in the event of an error.
- seed: an optional pseudo-random number generator (RNG) seed for each target. drake usually comes up with its own unique reproducible target-specific seeds using the global seed (the seed argument to make() and drake_config()) and the target names, but you can overwrite these automatic seeds. NA entries default back to drake's automatic seeds.
- max_expand: for dynamic branching only. Same as the max_expand argument of make(), but on a target-by-target basis. Limits the number of sub-targets created for a given target.

Formats

Specialized target formats increase efficiency and flexibility. Some allow you to save specialized objects like keras models, while others increase the speed while conserving storage and memory. You can declare target-specific formats in the plan (e.g. drake_plan(x = target(big_data_frame, format = "fst"))) or supply a global default format for all targets in make(). Either way, most formats have specialized installation requirements (e.g. R packages) that are not installed with drake by default. You will need to install them separately yourself. Available formats:

- "file": Dynamic files. To use this format, simply create local files and directories yourself and then return a character vector of paths as the target's value. Then, drake will watch for changes to those files in subsequent calls to make(). This is a more flexible alternative to file_in() and file_out(), and it is compatible with dynamic branching. See https://github.com/ropensci/drake/pull/1178 for an example.
- "fst": save big data frames fast. Requires the fst package. Note: this format strips non-data-frame attributes such as the
- "fst_tbl": Like "fst", but for tibble objects. Requires the fst and tibble packages. Strips away non-data-frame non-tibble attributes.
- "fst_dt": Like "fst" format, but for data.table objects. Requires the fst and data.table packages. Strips away non-data-frame non-data-table attributes.
- "diskframe": Stores disk.frame objects, which could potentially be larger than memory. Requires the fst and disk.frame packages. Coerces objects to disk.frames. Note: disk.frame objects get moved to the drake cache (a subfolder of .drake/ for most workflows). To ensure this data transfer is fast, it is best to save your disk.frame objects to the same physical storage drive as the drake cache, as.disk.frame(your_dataset,outdir = drake_tempfile()).
- "keras": save Keras models as HDF5 files. Requires the keras package.
- "qs": save any R object that can be properly serialized with the qs package. Requires the qs package. Uses qsave() and qread(). Uses the default settings in qs version 0.20.2.
- "rds": save any R object that can be properly serialized. Requires R version >= 3.5.0 due to ALTREP. Note: the "rds" format uses gzip compression, which is slow. "qs" is a superior format.

Keywords

drake_plan() understands special keyword functions for your commands. With the exception of target(), each one is a proper function with its own help file.

- target(): give the target more than just a command. Using target(), you can apply a transformation (examples: https://books.ropensci.org/drake/plans.html#large-plans), # nolint supply a trigger (https://books.ropensci.org/drake/triggers.html), # nolint or set any number of custom columns.
- file_in(): declare an input file dependency.
- file_out(): declare an output file to be produced when the target is built.
- knitr_in(): declare a knitr file dependency such as an R Markdown (*.Rmd) or R LaTeX (*.Rnw) file.
- ignore(): force drake to entirely ignore a piece of code: do not track it for changes and do not analyze it for dependencies.
- no_deps(): tell drake to not track the dependencies of a piece of code. drake still tracks the code itself for changes.
- id_chr(): Get the name of the current target.
- drake_envir(): get the environment where drake builds targets. Intended for advanced custom memory management.

Transformations

drake has special syntax for generating large plans. Your code will look something like drake_plan(y = target(f(x), transform = map(x = c(1, 2, 3))) You can read about this interface at https://books.ropensci.org/drake/plans.html#large-plans. # nolint

Static branching

In static branching, you define batches of targets based on information you know in advance. Overall usage looks like drake_plan(<x> = target(<...>, transform = <call>), where

- <x> is the name of the target or group of targets.
- <...> is optional arguments to target().
- <call> is a call to one of the transformation functions.

Transformation function usage:

- map(...,.data,.names,.id,.tag_in,.tag_out)
- split(...,slices,margin = 1L,drop = FALSE,.names,.tag_in,.tag_out) # nolint
- cross(...,.data,.names,.id,.tag_in,.tag_out)
- combine(...,.by,.names,.id,.tag_in,.tag_out)

Dynamic branching

- map(...,.trace)
- cross(...,.trace)
- group(...,.by,.trace)

map() and cross() create dynamic sub-targets from the variables supplied to the dots. As with static branching, the variables supplied to map() must all have equal length. group(f(data),.by = x) makes new dynamic sub-targets from data. Here, data can be either static or dynamic. If data is dynamic, group() aggregates existing sub-targets. If data is static, group() splits data into multiple subsets based on the groupings from .by.

Differences from static branching:

- ... must contain *unnamed* symbols with no values supplied, and they must be the names of targets.
- Arguments .id, .tag_in, and .tag_out no longer apply.

See Also

make, drake_config, transform_plan, map, split, cross, combine

Examples

```
## Not run:
isolate_example("contain side effects", {
# For more examples, visit
# https://books.ropensci.org/drake/plans.html.
```

```
# Create drake plans:
mtcars_plan <- drake_plan(
  write.csv(mtcars[, c("mpg", "cyl")], file_out("mtcars.csv")),
  value = read.csv(file_in("mtcars.csv"))
)
if (requireNamespace("visNetwork", quietly = TRUE)) {
  plot(mtcars_plan) # fast simplified call to vis_drake_graph()
}
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)
if (requireNamespace("knitr", quietly = TRUE)) {
  plot(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")

# Formats are great for big data: https://github.com/ropensci/drake/pull/977
# Below, each target is 1.6 GB in memory.
# Run make() on this plan to see how much faster fst is!
n <- 1e8
plan <- drake_plan(
  data_fst = target(
    data.frame(x = runif(n), y = runif(n)),
    format = "fst"
  ),
  data_old = data.frame(x = runif(n), y = runif(n))
)

# Use transformations to generate large plans.
# Read more at
# <https://books.ropensci.org/drake/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))
  )
)

# Split data among multiple targets.
drake_plan(
  large_data = get_data(),
  slice_analysis = target(
    analyze(large_data),
    transform = split(large_data, slices = 4)
  ),
  results = target(
    rbind(slice_analysis),
    transform = combine(slice_analysis)
  )
)

# 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)))

# Dynamic branching
# Get the mean mpg for each cyl in the mtcars dataset.
plan <- drake_plan(
  raw = mtcars,
  group_index = raw$cyl,
  munged = target(raw[, c("mpg", "cyl")], dynamic = map(raw)),
  mean_mpg_by_cyl = target(
    data.frame(mpg = mean(munged$mpg), cyl = munged$cyl[1]),
    dynamic = group(munged, .by = group_index)
  )
)
make(plan)
readd(mean_mpg_by_cyl)
})

## End(Not run)
```

`drake_plan_source` *Show the code required to produce a given* `drake` *plan* **Stable**

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.
}
```

---

drake_progress *Get the build progress of your targets* **Stable**

Description

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

Usage

```
drake_progress(
  ...,
  list = character(0),
  cache = drake::drake_cache(path = path),
  path = NULL,
  progress = NULL
)
```

Arguments

`...`	Objects to load from the cache, as names (unquoted) or character strings (quoted). 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 objects to be loaded from the cache. Similar to the `list` argument of `remove()`.
`cache`	drake cache. See `new_cache()`. If supplied, `path` is ignored.

path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.
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", and "failed".

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:
isolate_example("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 drake_progress() as make() is running.
drake_progress() # List all the targets reached so far.
drake_progress(small, large) # Just see the progress of some targets.
drake_progress(list = c("small", "large")) # Same as above.
}
})

## End(Not run)
```

---

drake_running *List running targets.* **Stable**

Description

List the targets that either

1. Are currently being built during a call to make(), or
2. Were in progress when make() was interrupted.

Usage

```
drake_running(cache = drake::drake_cache(path = path), path = NULL)
```

Arguments

cache	drake cache. See new_cache(). If supplied, path is ignored.
path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.

Value

A character vector of target names.

See Also

drake_done(), drake_failed(), drake_cancelled(), drake_progress(), make()

Examples

```
## Not run:
isolate_example("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_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.
# drake_running()
# nolint end
}
})

## End(Not run)
```

---

drake_script *Write an example* _drake.R *script to the current working directory.*

Description

A _drake.R file is required for r_make() and friends. See the r_make() help file for details.

Usage

```
drake_script(code = NULL)
```

Arguments

code	R code to put in _drake.R in the current working directory. If NULL, an example script is written.

Value

Nothing.

Examples

```
## Not run:
isolate_example("contain side-effects", {
drake_script({
  library(drake)
  plan <- drake_plan(x = 1)
  drake_config(plan, lock_cache = FALSE)
})
cat(readLines("_drake.R"), sep = "\n")
r_make()
})

## End(Not run)
```

---

drake_slice *Take a strategic subset of a dataset.* **Stable**

Description

drake_slice() is similar to split(). Both functions partition data into disjoint subsets, but whereas split() returns *all* the subsets, drake_slice() returns just *one*. In other words, drake_slice(...,index = i) returns split(...)[[i]]. Other features: 1. drake_slice() works on vectors, data frames, matrices, lists, and arbitrary arrays. 2. Like parallel::splitIndices(), drake_slice() tries to distribute the data uniformly across subsets. See the examples to learn why splitting is useful in drake.

Usage

```
drake_slice(data, slices, index, margin = 1L, drop = FALSE)
```

Arguments

data	A list, vector, data frame, matrix, or arbitrary array. Anything with a length() or dim().
slices	Integer of length 1, number of slices (i.e. pieces) of the whole dataset. Remember, drake_slice(index = i) returns only slice number i.
index	Integer of length 1, which piece of the partition to return.
margin	Integer of length 1, margin over which to split the data. For example, for a data frame or matrix, use margin = 1 to split over rows and margin = 2 to split over columns. Similar to MARGIN in apply().
drop	Logical, for matrices and arrays. If TRUE, the result is coerced to the lowest possible dimension. See ?[‘ for details.

Value

A subset of data.

Examples

```
# Simple usage
x <- matrix(seq_len(20), nrow = 5)
x
drake_slice(x, slices = 3, index = 1)
drake_slice(x, slices = 3, index = 2)
drake_slice(x, slices = 3, index = 3)
drake_slice(x, slices = 3, margin = 2, index = 1)
# In drake, you can split a large dataset over multiple targets.
## Not run:
isolate_example("contain side effects", {
plan <- drake_plan(
  large_data = mtcars,
  data_split = target(
    drake_slice(large_data, slices = 32, index = i),
    transform = map(i = !!seq_len(32))
  )
)
plan
cache <- storr::storr_environment()
make(plan, cache = cache, session_info = FALSE, verbose = FALSE)
readd(data_split_1L, cache = cache)
readd(data_split_2L, cache = cache)
})

## End(Not run)
```

---

drake_tempfile *drake tempfile* **Stable**

Description

Create the path to a temporary file inside drake's cache.

Usage

```
drake_tempfile(path = NULL, cache = drake::drake_cache(path = path))
```

Arguments

path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.
cache	drake cache. See new_cache(). If supplied, path is ignored.

Details

This function is just like the tempfile() function in base R except that the path points to a special location inside drake's cache. This ensures that if the file needs to be copied to persistent storage in the cache, drake does not need to copy across physical storage media. Example: the "diskframe" format. See the "Formats" and "Columns" sections of the drake_plan() help file. Unless you

supply the cache or the path to the cache (see drake_cache()) drake will assume the cache folder is named .drake/ and it is located either in your working directory or an ancestor of your working directory.

See Also

drake_cache(), new_cache()

Examples

```
cache <- new_cache(tempfile())
# No need to supply a cache if a .drake/ folder exists.
drake_tempfile(cache = cache)
drake_plan(
  x = target(
    as.disk.frame(large_data, outdir = drake_tempfile()),
    format = "diskframe"
  )
)
```

---

file_in *Declare input files and directories.* **Stable**

Description

file_in() marks individual files (and whole directories) that your targets depend on.

Usage

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

Arguments

...	Character vector, paths to files and directories. Use .id_chr to refer to the current target by name. .id_chr is not limited to use in file_in() and file_out().

Value

A character vector of declared input file or directory paths.

URLs

As of drake 7.4.0, file_in() and file_out() have support for URLs. If the file name begins with "http://", "https://", or "ftp://", make() attempts to check the ETag to see if the data changed from last time. If no ETag can be found, drake simply uses the ETag from last make() and registers the file as unchanged (which prevents your workflow from breaking if you lose internet access). If your file_in() URLs require authentication, see the curl_handles argument of make() and drake_config() to learn how to supply credentials.

Keywords

drake_plan() understands special keyword functions for your commands. With the exception of target(), each one is a proper function with its own help file.

- target(): give the target more than just a command. Using target(), you can apply a transformation (examples: https://books.ropensci.org/drake/plans.html#large-plans), # nolint supply a trigger (https://books.ropensci.org/drake/triggers.html), # nolint or set any number of custom columns.
- file_in(): declare an input file dependency.
- file_out(): declare an output file to be produced when the target is built.
- knitr_in(): declare a knitr file dependency such as an R Markdown (*.Rmd) or R LaTeX (*.Rnw) file.
- ignore(): force drake to entirely ignore a piece of code: do not track it for changes and do not analyze it for dependencies.
- no_deps(): tell drake to not track the dependencies of a piece of code. drake still tracks the code itself for changes.
- id_chr(): Get the name of the current target.
- drake_envir(): get the environment where drake builds targets. Intended for advanced custom memory management.

See Also

file_out(), knitr_in(), ignore(), no_deps()

Examples

```
## Not run:
isolate_example("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.
plan <- drake_plan(
  out = 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")

# You may use `.id_chr` inside `file_out()` and `file_in()`
# to refer  to the current target. This works inside
# static `map()`, `combine()`, `split()`, and `cross()`.

plan <- drake::drake_plan(
```

```
  data = target(
    write.csv(data, file_out(paste0(.id_chr, ".csv"))),
    transform = map(data = c(airquality, mtcars))
  )
)
plan

# You can also work with entire directories this way.
# However, in `file_out("your_directory")`, the directory
# becomes an entire unit. Thus, `file_in("your_directory")`
# is more appropriate for subsequent steps than
# `file_in("your_directory/file_inside.txt")`.
plan <- drake_plan(
  out = {
    dir.create(file_out("dir"))
    write.csv(mtcars, "dir/mtcars.csv")
  },
  contents = read.csv(file.path(file_in("dir"), "mtcars.csv"))
)
plan

make(plan)
file.exists("dir/mtcars.csv")

# See the connections that the file relationships create:
if (requireNamespace("visNetwork", quietly = TRUE)) {
  vis_drake_graph(plan)
}
})

## End(Not run)
```

`file_out` *Declare output files and directories.* **Stable**

Description

`file_out()` marks individual files (and whole directories) that your targets create.

Usage

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

Arguments

`...`	Character vector, paths to files and directories. Use `.id_chr` to refer to the current target by name. `.id_chr` is not limited to use in `file_in()` and `file_out()`.

Value

A character vector of declared output file or directory paths.

Keywords

drake_plan() understands special keyword functions for your commands. With the exception of target(), each one is a proper function with its own help file.

- target(): give the target more than just a command. Using target(), you can apply a transformation (examples: https://books.ropensci.org/drake/plans.html#large-plans), # nolint supply a trigger (https://books.ropensci.org/drake/triggers.html), # nolint or set any number of custom columns.
- file_in(): declare an input file dependency.
- file_out(): declare an output file to be produced when the target is built.
- knitr_in(): declare a knitr file dependency such as an R Markdown (*.Rmd) or R LaTeX (*.Rnw) file.
- ignore(): force drake to entirely ignore a piece of code: do not track it for changes and do not analyze it for dependencies.
- no_deps(): tell drake to not track the dependencies of a piece of code. drake still tracks the code itself for changes.
- id_chr(): Get the name of the current target.
- drake_envir(): get the environment where drake builds targets. Intended for advanced custom memory management.

See Also

file_in(), knitr_in(), ignore(), no_deps()

Examples

```
## Not run:
isolate_example("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.
plan <- drake_plan(
  out = 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")

 # You may use `.id_chr` inside `file_out()` and `file_in()`
 # to refer  to the current target. This works inside `map()`,
 # `combine()`, `split()`, and `cross()`.

plan <- drake::drake_plan(
```

```
  data = target(
    write.csv(data, file_out(paste0(.id_chr, ".csv"))),
    transform = map(data = c(airquality, mtcars))
  )
)

plan

# You can also work with entire directories this way.
# However, in `file_out("your_directory")`, the directory
# becomes an entire unit. Thus, `file_in("your_directory")`
# is more appropriate for subsequent steps than
# `file_in("your_directory/file_inside.txt")`.
plan <- drake_plan(
  out = {
    dir.create(file_out("dir"))
    write.csv(mtcars, "dir/mtcars.csv")
  },
  contents = read.csv(file.path(file_in("dir"), "mtcars.csv"))
)
plan

make(plan)
file.exists("dir/mtcars.csv")

# See the connections that the file relationships create:
if (requireNamespace("visNetwork", quietly = TRUE)) {
  vis_drake_graph(plan)
}
})

## End(Not run)
```

`file_store` *Show a file's encoded representation in the cache* **Stable**

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:
isolate_example("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"),
  plan = my_plan,
  character_only = TRUE
)
}
})

## End(Not run)
```

---

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

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:
isolate_example("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)
```

---

id_chr *Name of the current target* **Stable**

Description

id_chr() gives you the name of the current target while make() is running. For static branching in drake_plan(), use the .id_chr symbol instead. See the examples for details.

Usage

```
id_chr()
```

Value

The name of the current target.

Keywords

drake_plan() understands special keyword functions for your commands. With the exception of target(), each one is a proper function with its own help file.

- target(): give the target more than just a command. Using target(), you can apply a transformation (examples: https://books.ropensci.org/drake/plans.html#large-plans), # nolint supply a trigger (https://books.ropensci.org/drake/triggers.html), # nolint or set any number of custom columns.
- file_in(): declare an input file dependency.
- file_out(): declare an output file to be produced when the target is built.
- knitr_in(): declare a knitr file dependency such as an R Markdown (*.Rmd) or R LaTeX (*.Rnw) file.

- ignore(): force drake to entirely ignore a piece of code: do not track it for changes and do not analyze it for dependencies.
- no_deps(): tell drake to not track the dependencies of a piece of code. drake still tracks the code itself for changes.
- id_chr(): Get the name of the current target.
- drake_envir(): get the environment where drake builds targets. Intended for advanced custom memory management.

Examples

```
try(id_chr()) # Do not use outside the plan.
## Not run:
isolate_example("id_chr()", {
plan <- drake_plan(x = id_chr())
make(plan)
readd(x)
# Dynamic branching
plan <- drake_plan(
  x = seq_len(4),
  y = target(id_chr(), dynamic = map(x))
)
make(plan)
readd(y, subtargets = 1)
# Static branching
plan <- drake_plan(
  y = target(c(x, .id_chr), transform = map(x = !!seq_len(4)))
)
plan
})

## End(Not run)
```

---

ignore *Ignore code* **Stable**

Description

Ignore sections of commands and imported functions.

Usage

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

Arguments

x	Code to ignore.

Details

In user-defined functions and drake_plan() commands, you can wrap code chunks in ignore() to

1. Tell drake to not search for dependencies (targets etc. mentioned in the code) and
2. Ignore changes to the code so downstream targets remain up to date. To enforce (1) without (2), use no_deps().

Value

The argument.

Keywords

drake_plan() understands special keyword functions for your commands. With the exception of target(), each one is a proper function with its own help file.

- target(): give the target more than just a command. Using target(), you can apply a transformation (examples: https://books.ropensci.org/drake/plans.html#large-plans), # nolint supply a trigger (https://books.ropensci.org/drake/triggers.html), # nolint or set any number of custom columns.
- file_in(): declare an input file dependency.
- file_out(): declare an output file to be produced when the target is built.
- knitr_in(): declare a knitr file dependency such as an R Markdown (*.Rmd) or R LaTeX (*.Rnw) file.
- ignore(): force drake to entirely ignore a piece of code: do not track it for changes and do not analyze it for dependencies.
- no_deps(): tell drake to not track the dependencies of a piece of code. drake still tracks the code itself for changes.
- id_chr(): Get the name of the current target.
- drake_envir(): get the environment where drake builds targets. Intended for advanced custom memory management.

See Also

file_in(), file_out(), knitr_in(), no_deps()

Examples

```
## Not run:
isolate_example("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.

# ignore() works with functions and multiline code chunks.
f <- function(x) {
  ignore({
    x <- x + 1
    x <- x + 2
  })
  x # Not ignored.
}
make(plan = drake_plan(y = f(2)))
readd(x)
# Changes the content of the ignore() block:
f <- function(x) {
  ignore({
    x <- x + 1
  })
  x # Not ignored.
}
make(plan = drake_plan(x = f(2)))
readd(x)
})

## End(Not run)
```

`knitr_in` *Declare* `knitr`/`rmarkdown` *source files as dependencies.* **Stable**

Description

`knitr_in()` marks individual `knitr`/R Markdown reports as dependencies. In `drake`, these reports are pieces of the pipeline. R Markdown is a great tool for *displaying* precomputed results, but not for running a large workflow from end to end. These reports should do as little computation as possible.

Usage

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

Arguments

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

Details

Unlike `file_in()` and `file_out()`, `knitr_in()` does not work with entire directories.

Value

A character vector of declared input file paths.

Keywords

drake_plan() understands special keyword functions for your commands. With the exception of target(), each one is a proper function with its own help file.

- target(): give the target more than just a command. Using target(), you can apply a transformation (examples: https://books.ropensci.org/drake/plans.html#large-plans), # nolint supply a trigger (https://books.ropensci.org/drake/triggers.html), # nolint or set any number of custom columns.
- file_in(): declare an input file dependency.
- file_out(): declare an output file to be produced when the target is built.
- knitr_in(): declare a knitr file dependency such as an R Markdown (*.Rmd) or R LaTeX (*.Rnw) file.
- ignore(): force drake to entirely ignore a piece of code: do not track it for changes and do not analyze it for dependencies.
- no_deps(): tell drake to not track the dependencies of a piece of code. drake still tracks the code itself for changes.
- id_chr(): Get the name of the current target.
- drake_envir(): get the environment where drake builds targets. Intended for advanced custom memory management.

See Also

file_in(), file_out(), ignore(), no_deps()

Examples

```
## Not run:
isolate_example("contain side effects", {
if (requireNamespace("knitr", quietly = TRUE)) {
# `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.* **Soft-deprecated**

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.* **Stable**

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. This function also writes/overwrites the file, `report.Rmd`.

Value

Nothing.

See Also

`clean_mtcars_example()` `drake_examples()`

Examples

```
## Not run:
isolate_example("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.
outdated(my_plan) # Which targets are out of date?
# Run the workflow to build all the targets in the plan.
make(my_plan)
outdated(my_plan) # 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).* **Stable**

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. Because of how `make()` tracks global functions and objects as dependencies of targets, please restart your R session so the pipeline runs in a clean reproducible environment.

Usage

```
make(
  plan,
  targets = NULL,
  envir = parent.frame(),
  verbose = 1L,
  hook = NULL,
  cache = drake::drake_cache(),
  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 = NULL,
  cache_log_file = NULL,
  seed = NULL,
  caching = "main",
  keep_going = FALSE,
  session = NULL,
  pruning_strategy = NULL,
  makefile_path = NULL,
  console_log_file = NULL,
  ensure_workers = NULL,
  garbage_collection = FALSE,
  template = list(),
  sleep = function(i) 0.01,
  hasty_build = NULL,
  memory_strategy = "speed",
  layout = NULL,
  spec = NULL,
  lock_envir = TRUE,
  history = TRUE,
  recover = FALSE,
  recoverable = TRUE,
  curl_handles = list(),
  max_expand = NULL,
  log_build_times = TRUE,
  format = NULL,
  lock_cache = TRUE,
```

```
  log_make = NULL,
  log_worker = FALSE
)
```

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. You may supply static and/or whole dynamic targets, but no sub-targets.
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	Integer, control printing to the console/terminal. • 0: print nothing. • 1: print target-by-target messages as make() progresses. • 2: show a progress bar to track how many targets are done so far.
hook	Deprecated.
cache	drake cache as created by new_cache(). See also drake_cache().
fetch_cache	Deprecated.
parallelism	Character scalar, type of parallelism to use. For detailed explanations, see the high-performance computing chapter # nolint 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() 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://books.ropensci.org/drake/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 extra files in the cache (usually the .drake/ folder) and slows down `make()` a little. If you need to reduce or limit the number of files in the cache, call `make(log_progress = FALSE, recover = 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 — Deprecated.

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
: Deprecated.

lazy_load
: An old feature, currently being questioned. For the current recommendations on memory management, see https://books.ropensci.org/drake/memory.html#memory-strategies. The lazy_load argument is either a character vector or a logical. For dynamic targets, the behavior is always "eager" (see below). So the lazy_load argument is for static targets only. Choices for lazy_load:

  - "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".

  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. Defaults to TRUE. 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 CSV cache log file to write. If TRUE, the default file name is used (drake_cache.CSV). 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 exception to this last point is make(parallelism = "clustermq") 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, either "main" or "worker".

- "main": Targets are built by remote workers and sent back to the main process. Then, the main 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 Deprecated.

console_log_file
Deprecated in favor of log_make.

ensure_workers Deprecated.

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")), 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 Optional function on a single numeric argument i. Default: function(i) 0.01.

To conserve memory, drake assigns a brand new closure to sleep, so your custom function should not depend on in-memory data except from loaded packages.

For parallel processing, drake uses a central main 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 main process sleeps to avoid throttling. The sleep argument to make() and drake_config() allows you to customize how much time the main 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` Deprecated

`memory_strategy`

Character scalar, name of the strategy `drake` uses to load/unload a target's dependencies in memory. You can give each target its own memory strategy, (e.g. `drake_plan(x = 1,y = target(f(x),memory_strategy = "lookahead"))`) to override the global memory strategy. Choices:

- `"speed"`: Once a target is newly built or loaded in memory, just keep it there. This choice maximizes speed and hogs memory.
- `"autoclean"`: Just before building each new target, unload everything from memory except the target's direct dependencies. After a target is built, discard it from memory. (Set `garbage_collection = TRUE` to make sure it is really gone.) This option conserves memory, but it sacrifices speed because each new target needs to reload any previously unloaded targets from storage.
- `"preclean"`: Just before building each new target, unload everything from memory except the target's direct dependencies. After a target is built, keep it in memory until `drake` determines they can be unloaded. 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. After a target is built, keep it in memory until the next memory management stage. 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.
- `"unload"`: Just before building each new target, unload all targets from memory. After a target is built, **do not** keep it in memory. This mode aggressively optimizes for both memory and speed, but in commands and triggers, you have to manually load any dependencies you need using `readd()`.
- `"none"`: Do not manage memory at all. Do not load or unload anything before building targets. After a target is built, **do not** keep it in memory. This mode aggressively optimizes for both memory and speed, but in commands and triggers, you have to manually load any dependencies you need using `readd()`.

For even more direct control over which targets `drake` keeps in memory, see the help file examples of `drake_envir()`. Also see the `garbage_collection` argument of `make()` and `drake_config()`.

`layout` Deprecated.

`spec` Deprecated.

`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.

`history` Logical, whether to record the build history of your targets. You can also supply a `txtq`, which is how `drake` records history. Must be `TRUE` for `drake_history()` to work later.

`recover` Logical, whether to activate automated data recovery. The default is `FALSE` because

1. Automated data recovery is still stable.
2. It has reproducibility issues. Targets recovered from the distant past may have been generated with earlier versions of R and earlier package environments that no longer exist.
3. It is not always possible, especially when dynamic files are combined with dynamic branching (e.g. `dynamic = map(stuff)` and `format = "file"` etc.) since behavior is harder to predict in advance.

How it works: if `recover` is `TRUE`, `drake` tries to salvage old target values from the cache instead of running commands from the plan. A target is recoverable if

1. There is an old value somewhere in the cache that shares the command, dependencies, etc. of the target about to be built.
2. The old value was generated with `make(recoverable = TRUE)`.

If both conditions are met, `drake` will

1. Assign the most recently-generated admissible data to the target, and
2. skip the target's command.

Functions `recoverable()` and `r_recoverable()` show the most upstream outdated targets that will be recovered in this way in the next `make()` or `r_make()`.

`recoverable` Logical, whether to make target values recoverable with `make(recover = TRUE)`. This requires writing extra files to the cache, and it prevents old metadata from being removed with garbage collection (`clean(garbage_collection = TRUE)`, `gc()` in `storrs`). If you need to limit the cache size or the number of files in the cache, consider `make(recoverable = FALSE, progress = FALSE)`. Recovery is not always possible, especially when dynamic files are combined with dynamic branching (e.g. `dynamic = map(stuff)` and `format = "file"` etc.) since behavior is harder to predict in advance.

`curl_handles` A named list of curl handles. Each value is an object from `curl::new_handle()`, and each name is a URL (and should start with "http", "https", or "ftp"). Example: list(http://httpbin.org/basic-auth = curl::new_handle(username = "user", password = "passwd")) Then, if your plan has `file_in("http://httpbin.org/basic-auth/user/pas`
`drake` will authenticate using the username and password of the handle for http://httpbin.org/basic-auth/.

`drake` uses partial matching on text to find the right handle of the `file_in()` URL, so the name of the handle could be the complete URL (`"http://httpbin.org/basic-auth/user/`
or a part of the URL (e.g. `"http://httpbin.org/"` or `"http://httpbin.org/basic-auth/"`).
If you have multiple handles whose names match your URL, `drake` will choose the closest match.

max_expand
: Positive integer, optional. max_expand is the maximum number of targets to generate in each map(), cross(), or group() dynamic transform. Useful if you have a massive number of dynamic sub-targets and you want to work with only the first few sub-targets before scaling up. Note: the max_expand argument of make() and drake_config() is for dynamic branching only. The static branching max_expand is an argument of drake_plan() and transform_plan().

log_build_times
: Logical, whether to record build_times for targets. Mac users may notice a 20% speedup in make() with build_times = FALSE.

format
: Character, an optional custom storage format for targets without an explicit target(format = ...) in the plan. Details about formats: https://books.ropensci.org/drake/plans.html#special-data-formats-for-targets # nolint

lock_cache
: Logical, whether to lock the cache before running make() etc. It is usually recommended to keep cache locking on. However, if you interrupt make() before it can clean itself up, then the cache will stay locked, and you will need to manually unlock it with drake::drake_cache("xyz")$unlock(). Repeatedly unlocking the cache by hand is annoying, and lock_cache = FALSE prevents the cache from locking in the first place.

log_make
: Optional character scalar of a file name or connection object (such as stdout()) to dump maximally verbose log information for make() and other functions (all functions that accept a config argument, plus drake_config()). If you choose to use a text file as the console log, it will persist over multiple function calls until you delete it manually. Fields in each row the log file, from left to right: - The node name (short host name) of the computer (from Sys.info()["nodename"]). - The process ID (from Sys.getpid()). - A timestamp with the date and time (in microseconds). - A brief description of what drake was doing. The fields are separated by pipe symbols ("|"‘).

log_worker
: Logical, same as the log_worker argument of clustermq::workers() and clustermq::Q(). Only relevant if parallelism is "clustermq".

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. If you do run make() interactively, please restart your R session beforehand so your functions and global objects get loaded into a clean reproducible environment. This prevents targets from getting invalidated unexpectedly.

A serious drake workflow should be consistent and reliable, ideally with the help of a main 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/main/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.

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)
)
```

Here, because `data()` loads `mtcars` into the global environment, 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. Functions such as `data()` belong in your setup scripts prior to `make()`, not in any functions or commands that get called during `make()` itself.

For each target that is still problematic (e.g. https://github.com/rstudio/gt/issues/297) you can safely run the command in its own special `callr::r()` process. Example: https://github.com/rstudio/gt/issues/297#issuecomment-497778735. # nolint

If that fails, you can run `make(plan,lock_envir = FALSE)` to suppress environment-locking for all targets. However, this is not usually recommended. There are legitimate use cases for `lock_envir = FALSE` (example: https://books.ropensci.org/drake/hpc.html#parallel-computing-within-targets) # nolint but most workflows should stick with the default `lock_envir = TRUE`.

Cache locking

When `make()` runs, it locks the cache so other processes cannot modify it. Same goes for `outdated()`, `vis_drake_graph()`, and similar functions when `make_imports = TRUE`. This is a safety measure to prevent simultaneous processes from corrupting the cache. If you get an error saying that the cache is locked, either set `make_imports = FALSE` or manually force unlock it with `drake_cache()$unlock()`.

See Also

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

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
config <- drake_config(my_plan)
outdated(my_plan) # Which targets need to be (re)built?
make(my_plan) # Build what needs to be built.
outdated(my_plan) # 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(my_plan) # Some targets depend on reg2().
make(my_plan) # Rebuild just the outdated targets.
outdated(my_plan) # Everything is up to date again.
if (requireNamespace("visNetwork", quietly = TRUE)) {
vis_drake_graph(my_plan) # See how they fit in an interactive graph.
make(my_plan, cache_log_file = TRUE) # Write a CSV log file this time.
vis_drake_graph(my_plan) # The colors changed in the graph.
# Run targets in parallel:
# options(clustermq.scheduler = "multicore") # nolint
# make(my_plan, parallelism = "clustermq", jobs = 2) # nolint
}
clean() # Start from scratch next time around.
}
# Dynamic branching
# Get the mean mpg for each cyl in the mtcars dataset.
plan <- drake_plan(
  raw = mtcars,
  group_index = raw$cyl,
  munged = target(raw[, c("mpg", "cyl")], dynamic = map(raw)),
  mean_mpg_by_cyl = target(
    data.frame(mpg = mean(munged$mpg), cyl = munged$cyl[1]),
    dynamic = group(munged, .by = group_index)
  )
)
make(plan)
readd(mean_mpg_by_cyl)
})

## End(Not run)
```

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

Description

Checks your workspace/environment and file system.

Usage

```
missed(..., config = NULL)
```

Arguments

`...`	Arguments to `make()`, such as `plan` and `targets`.
`config`	Deprecated.

Value

Character vector of names of missing objects and files.

See Also

outdated()

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
plan <- drake_plan(x = missing::fun(arg))
missed(plan)
}
})

## End(Not run)
```

---

new_cache *Make a new* drake *cache.* **'Stable**

Description

Uses the storr_rds() function from the storr package.

Usage

```
new_cache(
  path = NULL,
  verbose = NULL,
  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	Deprecated on 2019-09-11.
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
: Deprecated on 2019-09-11.

Value

A newly created drake cache as a storr object.

See Also

make()

Examples

```
## Not run:
isolate_example("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)
```

---

no_deps *Suppress dependency detection.* **Stable**

Description

Tell drake to not search for dependencies in a chunk of code.

Usage

```
no_deps(x = NULL)
```

Arguments

x
: Code for which dependency detection is suppressed.

Details

no_deps() is similar to ignore(), but it still lets drake track meaningful changes to the code itself.

Value

The argument.

Keywords

drake_plan() understands special keyword functions for your commands. With the exception of target(), each one is a proper function with its own help file.

- target(): give the target more than just a command. Using target(), you can apply a transformation (examples: https://books.ropensci.org/drake/plans.html#large-plans), # nolint supply a trigger (https://books.ropensci.org/drake/triggers.html), # nolint or set any number of custom columns.
- file_in(): declare an input file dependency.
- file_out(): declare an output file to be produced when the target is built.
- knitr_in(): declare a knitr file dependency such as an R Markdown (*.Rmd) or R LaTeX (*.Rnw) file.
- ignore(): force drake to entirely ignore a piece of code: do not track it for changes and do not analyze it for dependencies.
- no_deps(): tell drake to not track the dependencies of a piece of code. drake still tracks the code itself for changes.
- id_chr(): Get the name of the current target.
- drake_envir(): get the environment where drake builds targets. Intended for advanced custom memory management.

See Also

file_in(), file_out(), knitr_in(), no_deps()

Examples

```
## Not run:
isolate_example("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 no_deps().
make(plan = drake_plan(y = sqrt(4) + no_deps(x))) # Builds y.
x <- 6
make(plan = drake_plan(y = sqrt(4) + no_deps(x))) # Skips y.
# However, `drake` *does* react to changes
# to the *literal code* inside `no_deps()`.
make(plan = drake_plan(y = sqrt(4) + ignore(x + 1))) # Builds y.

# Like ignore(), no_deps() works with functions and multiline code chunks.
z <- 1
```

```
f <- function(x) {
  no_deps({
    x <- z + 1
    x <- x + 2
  })
  x
}
make(plan = drake_plan(y = f(2)))
readd(y)
z <- 2 # Changed dependency is not tracked.
make(plan = drake_plan(y = f(2)))
readd(y)
})

## End(Not run)
```

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

Description

Outdated targets will be rebuilt in the next `make()`. `outdated()` does not show dynamic sub-targets.

Usage

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

Arguments

`...`	Arguments to `make()`, such as `plan` and `targets` and `envir`.
`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()`.
`config`	Deprecated (2019-12-21). A configured workflow from `drake_config()`.

Value

Character vector of the names of outdated targets.

See Also

`r_outdated()`, `drake_config()`, `missed()`, `drake_plan()`, `make()`

Examples

```
## Not run:
isolate_example("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.
outdated(my_plan) # Which targets are out of date?
make(my_plan) # Run the projects, build the targets.
# Now, everything should be up to date (no targets listed).
outdated(my_plan)
}
})

## End(Not run)
```

---

plan_to_code *Turn a* drake *plan into a plain R script file.* **Questioning**

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(Ozone ~ Temp + Wind, 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.
code_to_plan(file)
```

`plan_to_notebook` *Turn a* `drake` *plan into an R notebook.* **Questioning**

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 (suppressWarnings(require("knitr"))) {
plan <- drake_plan(
  raw_data = read_excel(file_in("raw_data.xlsx")),
  data = raw_data,
  hist = create_plot(data),
```

```
  fit = lm(Ozone ~ Temp + Wind, 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.* **Stable**

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(
  ...,
  targets_predict = NULL,
  from_scratch = FALSE,
  targets_only = NULL,
  jobs_predict = 1L,
  known_times = numeric(0),
  default_time = 0,
  warn = TRUE,
  config = NULL
)
```

Arguments

`...`	Arguments to `make()`, such as `plan` and `targets`.
`targets_predict`	Character vector, names of targets to include in the total runtime and worker predictions.
`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_predict`	The `jobs` argument of your next planned `make()`.

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.
config	Deprecated.

Value

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

See Also

predict_workers(), build_times(), make()

Examples

```
## Not run:
isolate_example("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.
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(
  my_plan,
  jobs_predict = 7L,
  from_scratch = TRUE,
  known_times = known_times
)
predict_runtime(
  my_plan,
  jobs_predict = 8L,
  from_scratch = TRUE,
  known_times = known_times
)
balance <- predict_workers(
  my_plan,
  jobs_predict = 7L,
  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.* **Stable**

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(
  ...,
  targets_predict = NULL,
  from_scratch = FALSE,
  targets_only = NULL,
  jobs_predict = 1L,
  known_times = numeric(0),
  default_time = 0,
  warn = TRUE,
  config = NULL
)
```

Arguments

`...`	Arguments to `make()`, such as `plan` and `targets`.
`targets_predict`	Character vector, names of targets to include in the total runtime and worker predictions.
`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_predict`	The `jobs` argument of your next planned `make()`.
`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`.
`config`	Deprecated.

Value

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

See Also

predict_runtime(), build_times(), make()

Examples

```
## Not run:
isolate_example("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.
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(
  my_plan,
  jobs_predict = 7L,
  from_scratch = TRUE,
  known_times = known_times
)
predict_runtime(
  my_plan,
  jobs_predict = 8L,
  from_scratch = TRUE,
  known_times = known_times
)
balance <- predict_workers(
  my_plan,
  jobs_predict = 7L,
  from_scratch = TRUE,
  known_times = known_times
)
balance
}
}
})

## End(Not run)
```

---

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

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(). If target is dynamic, readd() and loadd() retrieve a list of sub-target values. You can restrict which sub-targets to include using the subtargets argument.

Usage

```
readd(
  target,
  character_only = FALSE,
  path = NULL,
  search = NULL,
  cache = drake::drake_cache(path = path),
  namespace = NULL,
  verbose = 1L,
  show_source = FALSE,
  subtargets = NULL,
  subtarget_list = FALSE
)

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

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 — Path to a drake cache (usually a hidden .drake/ folder) or NULL.

search — Deprecated.

cache — drake cache. See new_cache(). If supplied, path is ignored.

namespace — Optional character string, name of the storr namespace to read from.

verbose — Deprecated on 2019-09-11.

show_source — Logical, option to show the command that produced the target or indicate that the object was imported (using show_source()).

subtargets — A numeric vector of indices. If target is dynamic, loadd() and readd() retrieve a list of sub-targets. You can restrict which sub-targets to retrieve with the subtargets argument. For example, readd(x, subtargets = seq_len(3)) only retrieves the first 3 sub-targets of dynamic target x.

subtarget_list — Logical, for dynamic targets only. If TRUE, the dynamic target is loaded as a named list of sub-target values. If FALSE, drake attempts to concatenate the sub-targets with vctrs::vec_c() (and returns an unnamed list if such concatenation is not possible).

... — 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.

Important note: deps = TRUE disables tidyselect functionality. For example, loadd(starts_with("model_"), config = config, deps = TRUE) does not work. For the selection mechanism to work, the model_* targets to need to already be in the cache, which is not always the case when you are debugging your projects. To help drake understand what you mean, you must name the targets *explicitly* when deps is TRUE, e.g. loadd(model_A, model_B, config = config, deps = 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 three 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.
3. If you are using make(memory_strategy = "none") or make(memory_strategy = "unload"), loadd() and readd() can manually load dependencies into memory for the target that is being built. If you do this, you must carefully inspect deps_target() and vis_drake_graph() before running make() to be sure the dependency relationships among targets are correct. If you do not wish to incur extra dependencies with loadd() or readd(), you will need to use ignore(), e.g. drake_plan(x = 1, y = ignore(readd(x))) or drake_plan(x = 1, y = readd(ignore("x"), character_only = TRUE)). Compare those plans to drake_plan(x = 1, y = readd(x)) and drake_plan(x = 1, y = readd("x", character_only = TRUE)) using vis_drake_graph() and deps_target().

Value

The cached value of the target.

See Also

cached(), drake_plan(), make()

cached(), drake_plan(), make()

Examples

```
## Not run:
isolate_example("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:
isolate_example("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.* **Stable**

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 = NULL, search = NULL, cache = NULL, verbose = NULL)
```

Arguments

path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.
search	Deprecated.
cache	drake cache. See new_cache(). If supplied, path is ignored.
verbose	Deprecated on 2019-09-11.

Value

An integer vector.

Examples

```
## Not run:
isolate_example("contain side effects", {
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)
})

## End(Not run)
```

read_trace *Read a trace of a dynamic target.* **Stable**

Description

Read a target's dynamic trace from the cache. Best used on its own outside a drake plan.

Usage

```
read_trace(
  trace,
  target,
  cache = drake::drake_cache(path = path),
  path = NULL,
  character_only = FALSE
)
```

Arguments

trace	Character, name of the trace you want to extract. Such trace names are declared in the .trace argument of map(), cross() or group().
target	Symbol or character, depending on the value of character_only. target is T=the name of a dynamic target with one or more traces defined using the .trace argument of dynamic map(), cross(), or group().
cache	drake cache. See new_cache(). If supplied, path is ignored.
path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.
character_only	Logical, whether name should be treated as a character or a symbol (just like character.only in library()).

Details

In dynamic branching, the trace keeps track of how the sub-targets were generated. It reminds us the values of grouping variables that go with individual sub-targets.

Value

The dynamic trace of one target in another: a vector of values from a grouping variable.

See Also

get_trace(), subtargets()

Examples

```
## Not run:
isolate_example("demonstrate dynamic trace", {
plan <- drake_plan(
  w = LETTERS[seq_len(3)],
  x = letters[seq_len(2)],

  # The first trace lets us see the values of w
  # that go with the sub-targets of y.
  y = target(paste0(w, x), dynamic = cross(w, x, .trace = w)),

  # We can use the trace as a grouping variable for the next
  # group().
  w_tr = read_trace("w", y),

  # Now, we use the trace again to keep track of the
  # values of w corresponding to the sub-targets of z.
  z = target(
    paste0(y, collapse = "-"),
    dynamic = group(y, .by = w_tr, .trace = w_tr)
  )
)
make(plan)

# We can read the trace outside make().
# That way, we know which values of `w` correspond
# to the sub-targets of `y`.
readd(y)
read_trace("w", y)

# And we know which values of `w_tr` (and thus `w`)
# match up with the sub-targets of `y`.
readd(z)
read_trace("w_tr", z)
})

## End(Not run)
```

`recoverable` *List the most upstream* recoverable *outdated targets.* **Stable**

Description

Only shows the most upstream updated targets. Whether downstream targets are recoverable depends on the eventual values of the upstream targets in the next `make()`.

Usage

```
recoverable(..., make_imports = TRUE, do_prework = TRUE, config = NULL)
```

Arguments

...	Arguments to make(), such as plan and targets and envir.
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().
config	Deprecated (2019-12-21). A configured workflow from drake_config().

Value

Character vector of the names of recoverable targets.

Recovery

make(recover = TRUE, recoverable = TRUE) powers automated data recovery. The default of recover is FALSE because targets recovered from the distant past may have been generated with earlier versions of R and earlier package environments that no longer exist.

How it works: if recover is TRUE, drake tries to salvage old target values from the cache instead of running commands from the plan. A target is recoverable if

1. There is an old value somewhere in the cache that shares the command, dependencies, etc. of the target about to be built.
2. The old value was generated with make(recoverable = TRUE).

If both conditions are met, drake will

1. Assign the most recently-generated admissible data to the target, and
2. skip the target's command.

See Also

r_recoverable(), r_outdated(), drake_config(), missed(), drake_plan(), make()

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
make(my_plan)
clean()
outdated(my_plan) # Which targets are outdated?
recoverable(my_plan) # Which of these are recoverable and upstream?
# The report still builds because clean() removes report.md,
# but make() recovers the rest.
make(my_plan, recover = TRUE)
outdated(my_plan)
# When was the *recovered* small data actually built (first stored)?
# (Was I using a different version of R back then?)
diagnose(small)$date
```

```
# If you set the same seed as before, you can even
# rename targets without having to build them again.
# For an example, see
# the "Reproducible data recovery and renaming" section of
# https://github.com/ropensci/drake/blob/main/README.md.
}
})

## End(Not run)
```

---

render_drake_ggraph *Visualize the workflow with* ggplot2/ggraph *using* drake_graph_info() *output.* **Stable**

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,
  label_nodes = FALSE,
  transparency = TRUE
)
```

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.
label_nodes	Logical, whether to label the nodes. If FALSE, the graph will not have any text next to the nodes, which is recommended for large graphs with lots of targets.
transparency	Logical, whether to allow transparency in the rendered graph. Set to FALSE if you get warnings like "semi-transparency is not supported on this device".

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:
isolate_example("Quarantine side effects.", {
load_mtcars_example() # Get the code with drake_example("mtcars").
if (requireNamespace("ggraph", quietly = TRUE)) {
  # Instead of jumping right to vis_drake_graph(), get the data frames
  # of nodes, edges, and legend nodes.
  drake_ggraph(my_plan) # Jump straight to the static graph.
  # Get the node and edge info that vis_drake_graph() just plotted:
  graph <- drake_graph_info(my_plan)
  render_drake_ggraph(graph)
}
})

## End(Not run)
```

---

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

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 = NULL,
  direction = NULL,
  hover = TRUE,
  main = graph_info$default_title,
  selfcontained = FALSE,
  navigationButtons = TRUE,
  ncol_legend = 1,
  collapse = TRUE,
  on_select = NULL,
  level_separation = NULL,
  ...
)
```

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_phantomjs(). 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	Deprecated.
direction	Deprecated.
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 = TRUE)
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).
on_select	defines node selection event handling. Either a string of valid JavaScript that may be passed to visNetwork::visEvents(), or one of the following: TRUE, NULL/FALSE. If TRUE , enables the default behavior of opening the link specified by the on_select_col given to drake_graph_info(). NULL/FALSE disables the behavior.
level_separation	Numeric, levelSeparation argument to visNetwork::visHierarchicalLayout(). Controls the distance between hierarchical levels. Consider setting if the aspect ratio of the graph is far from 1. Defaults to 150 through visNetwork.
...	Arguments passed to visNetwork().

Details

For enhanced interactivity in the graph, see the mandrake package: https://github.com/matthewstrasiotto/mandrake.

Value

A visNetwork graph.

See Also

vis_drake_graph(), sankey_drake_graph(), drake_ggraph()

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
if (requireNamespace("visNetwork", quietly = TRUE)) {
# Instead of jumping right to vis_drake_graph(), get the data frames
# of nodes, edges, and legend nodes.
vis_drake_graph(my_plan) # Jump straight to the interactive graph.
# Get the node and edge info that vis_drake_graph() just plotted:
graph <- drake_graph_info(my_plan)
# 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)
}
}
})

## End(Not run)
```

---

render_sankey_drake_graph

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

Description

This function is called inside sankey_drake_graph(), which typical users call more often. A legend is unfortunately unavailable for the graph itself, 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_phantomjs(). 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:
isolate_example("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.
sankey_drake_graph(my_plan) # 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(my_plan)
# 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)
}
}
}
})

## End(Not run)
```

render_text_drake_graph *Show a workflow graph as text in your terminal window using* drake_graph_info() *output.* **Stable**

Description

This function is called inside text_drake_graph(), which typical users call more often. See ?text_drake_graph for details.

Usage

```
render_text_drake_graph(graph_info, nchar = 1L, print = TRUE)
```

Arguments

graph_info	List of data frames generated by drake_graph_info(). There should be 3 data frames: nodes, edges, and legend_nodes.
nchar	For each node, maximum number of characters of the node label to show. Can be 0, in which case each node is a colored box instead of a node label. Caution: nchar > 0 will mess with the layout.
print	Logical. If TRUE, the graph will print to the console via message(). If FALSE, nothing is printed. However, you still have the visualization because text_drake_graph() and render_text_drake_graph() still invisibly return a character string that you can print yourself with message().

Value

The lines of text in the visualization.

See Also

text_drake_graph(), vis_drake_graph(), sankey_drake_graph(), drake_ggraph()

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
pkgs <- requireNamespace("txtplot", quietly = TRUE) &&
  requireNamespace("visNetwork", quietly = TRUE)
if (pkgs) {
# Instead of jumpting right to vis_drake_graph(), get the data frames
# of nodes, edges, and legend nodes.
text_drake_graph(my_plan) # Jump straight to the interactive graph.
# Get the node and edge info that vis_drake_graph() just plotted:
graph <- drake_graph_info(my_plan)
```

```
# You can pass the data frames right to render_text_drake_graph().
render_text_drake_graph(graph)
}
}
})

## End(Not run)
```

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

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 = NULL,
  search = NULL,
  verbose = NULL,
  force = FALSE,
  cache = drake::drake_cache(path = path),
  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. Set `path` to the path of a `storr::storr_rds()` cache to retrieve a specific cache generated by `storr::storr_rds()` or `drake::new_cache()`. If the `path` argument is `NULL`, `drake_cache()` searches up through parent directories to find a folder called .drake/.
`search`	Deprecated.
`verbose`	Deprecated on 2019-09-11.
`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

Nothing.

See Also

drake_cache(), cached(), drake_gc(), clean()

Examples

```
## Not run:
isolate_example("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)
```

---

r_make *Launch a drake function in a fresh new R process* **Stable**

Description

The r_*() functions, such as r_make(), enhance reproducibility by launching a drake function in a separate R process.

Usage

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

r_drake_build(
  target,
  character_only = FALSE,
  ...,
  source = NULL,
  r_fn = NULL,
  r_args = list()
)

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

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

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

r_deps_target(
  target,
  character_only = FALSE,
  ...,
  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_text_drake_graph(..., 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 = list(stdout = "stdout.log")).
target	Name of the target.
character_only	Logical, whether name should be treated as a character or a symbol (just like character.only in library()).
...	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

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.

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 calls drake_config() at the very end. drake_config() is the preprocessing step of make(), and it accepts all the same arguments as make() (e.g. plan and targets).
3. In that same session, run outdated() with the config argument from step 2.
4. Return the result back to main process (e.g. your interactive R session).

Recovery

make(recover = TRUE,recoverable = TRUE) powers automated data recovery. The default of recover is FALSE because targets recovered from the distant past may have been generated with earlier versions of R and earlier package environments that no longer exist.

How it works: if recover is TRUE, drake tries to salvage old target values from the cache instead of running commands from the plan. A target is recoverable if

1. There is an old value somewhere in the cache that shares the command, dependencies, etc. of the target about to be built.
2. The old value was generated with make(recoverable = TRUE).

If both conditions are met, drake will

1. Assign the most recently-generated admissible data to the target, and
2. skip the target's command.

See Also

make()

Examples

```
## Not run:
isolate_example("quarantine side effects", {
if (requireNamespace("knitr", quietly = TRUE)) {
writeLines(
  c(
    "library(drake)",
    "load_mtcars_example()",
    "drake_config(my_plan, targets = c(\"small\", \"large\"))"
  ),
  "_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.* **Stable**

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, but you can see what all the colors mean with visNetwork::visNetwork(drake::legend_nodes()).

Usage

```
sankey_drake_graph(
  ...,
  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,
  config = NULL
)
```

Arguments

...	Arguments to make(), such as plan and targets.
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_phantomjs(). 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 original drake 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.

config — Deprecated.

Value

A visNetwork graph.

See Also

render_sankey_drake_graph(), vis_drake_graph(), drake_ggraph(), text_drake_graph()

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
if (requireNamespace("networkD3", quietly = TRUE)) {
if (requireNamespace("visNetwork", quietly = TRUE)) {
# Plot the network graph representation of the workflow.
sankey_drake_graph(my_plan)
# Show the legend separately.
visNetwork::visNetwork(nodes = drake::legend_nodes())
make(my_plan) # Run the project, build the targets.
sankey_drake_graph(my_plan) # The black nodes from before are now green.
# Plot a subgraph of the workflow.
sankey_drake_graph(my_plan, from = c("small", "reg2"))
}
}
}
})

## End(Not run)
```

---

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

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

```
## Not run:
isolate_example("contain side effects", {
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, history = FALSE)
show_source(x, config)
})

## End(Not run)
```

---

subtargets *List sub-targets* **Stable**

Description

List the sub-targets of a dynamic target.

Usage

```
subtargets(
  target = NULL,
  character_only = FALSE,
  cache = drake::drake_cache(path = path),
  path = NULL
)
```

Arguments

target	Character string or symbol, depending on character_only. Name of a dynamic target.
character_only	Logical, whether target should be treated as a character or a symbol. Just like character.only in library().
cache	drake cache. See new_cache(). If supplied, path is ignored.
path	Path to a drake cache (usually a hidden .drake/ folder) or NULL.

Value

Character vector of sub-target names

See Also

get_trace(), read_trace()

Examples

```
## Not run:
isolate_example("dynamic branching", {
plan <- drake_plan(
  w = c("a", "a", "b", "b"),
  x = seq_len(4),
  y = target(x + 1, dynamic = map(x)),
  z = target(sum(x) + sum(y), dynamic = group(x, y, .by = w))
)
make(plan)
subtargets(y)
subtargets(z)
readd(x)
readd(y)
readd(z)
})

## End(Not run)
```

`target` *Customize a target in* `drake_plan()`. **Stable**

Description

The `target()` function is a way to configure individual targets in a `drake` plan. Its most common use is to invoke static branching and dynamic branching, and it can also set the values of custom columns such as `format`, `elapsed`, `retries`, and `max_expand`. Details are at https://books.ropensci.org/drake/plans.html#special-columns. Note: `drake_plan(my_target = my_command())` is equivalent to drake_plan(my_target = target(my_command())).

Usage

```
target(command = NULL, transform = NULL, dynamic = NULL, ...)
```

Arguments

`command`	The command to build the target.
`transform`	A call to `map()`, `split()`, `cross()`, or `combine()` to apply a *static* transformation. Details: https://books.ropensci.org/drake/static.html
`dynamic`	A call to `map()`, `cross()`, or `group()` to apply a *dynamic* transformation. Details: https://books.ropensci.org/drake/dynamic.html
`...`	Optional columns of the plan for a given target. See the Columns section of this help file for a selection of special columns that `drake` understands.

Details

`target()` must be called inside `drake_plan()`. It is invalid otherwise.

Value

A one-row workflow plan data frame with the named arguments as columns.

Columns

drake_plan() creates a special data frame. At minimum, that data frame must have columns target and command with the target names and the R code chunks to build them, respectively.

You can add custom columns yourself, either with target() (e.g. drake_plan(y = target(f(x), transform = map(c(1, 2)), format = "fst"))) or by appending columns post-hoc (e.g. plan$col <- vals).

Some of these custom columns are special. They are optional, but drake looks for them at various points in the workflow.

- transform: a call to map(), split(), cross(), or combine() to create and manipulate large collections of targets. Details: (https://books.ropensci.org/drake/plans.html#large-plans). # nolint
- format: set a storage format to save big targets more efficiently. See the "Formats" section of this help file for more details.
- trigger: rule to decide whether a target needs to run. It is recommended that you define this one with target(). Details: https://books.ropensci.org/drake/triggers.html.
- hpc: logical values (TRUE/FALSE/NA) whether to send each target to parallel workers. Visit https://books.ropensci.org/drake/hpc.html#selectivity to learn more.
- resources: target-specific lists of resources for a computing cluster. See https://books.ropensci.org/drake/hpc.html#advanced-options for details.
- caching: overrides the caching argument of make() for each target individually. Possible values:
  - "main": tell the main process to store the target in the cache.
  - "worker": tell the HPC worker to store the target in the cache.
  - NA: default to the caching argument of make().
- elapsed and cpu: number of seconds to wait for the target to build before timing out (elapsed for elapsed time and cpu for CPU time).
- retries: number of times to retry building a target in the event of an error.
- seed: an optional pseudo-random number generator (RNG) seed for each target. drake usually comes up with its own unique reproducible target-specific seeds using the global seed (the seed argument to make() and drake_config()) and the target names, but you can overwrite these automatic seeds. NA entries default back to drake's automatic seeds.
- max_expand: for dynamic branching only. Same as the max_expand argument of make(), but on a target-by-target basis. Limits the number of sub-targets created for a given target.

Keywords

drake_plan() understands special keyword functions for your commands. With the exception of target(), each one is a proper function with its own help file.

- target(): give the target more than just a command. Using target(), you can apply a transformation (examples: https://books.ropensci.org/drake/plans.html#large-plans), # nolint supply a trigger (https://books.ropensci.org/drake/triggers.html), # nolint or set any number of custom columns.
- file_in(): declare an input file dependency.
- file_out(): declare an output file to be produced when the target is built.
- knitr_in(): declare a knitr file dependency such as an R Markdown (*.Rmd) or R LaTeX (*.Rnw) file.
- ignore(): force drake to entirely ignore a piece of code: do not track it for changes and do not analyze it for dependencies.
- no_deps(): tell drake to not track the dependencies of a piece of code. drake still tracks the code itself for changes.
- id_chr(): Get the name of the current target.
- drake_envir(): get the environment where drake builds targets. Intended for advanced custom memory management.

Formats

Specialized target formats increase efficiency and flexibility. Some allow you to save specialized objects like keras models, while others increase the speed while conserving storage and memory. You can declare target-specific formats in the plan (e.g. drake_plan(x = target(big_data_frame, format = "fst"))) or supply a global default format for all targets in make(). Either way, most formats have specialized installation requirements (e.g. R packages) that are not installed with drake by default. You will need to install them separately yourself. Available formats:

- "file": Dynamic files. To use this format, simply create local files and directories yourself and then return a character vector of paths as the target's value. Then, drake will watch for changes to those files in subsequent calls to make(). This is a more flexible alternative to file_in() and file_out(), and it is compatible with dynamic branching. See https://github.com/ropensci/drake/pull/1178 for an example.
- "fst": save big data frames fast. Requires the fst package. Note: this format strips non-data-frame attributes such as the
- "fst_tbl": Like "fst", but for tibble objects. Requires the fst and tibble packages. Strips away non-data-frame non-tibble attributes.
- "fst_dt": Like "fst" format, but for data.table objects. Requires the fst and data.table packages. Strips away non-data-frame non-data-table attributes.
- "diskframe": Stores disk.frame objects, which could potentially be larger than memory. Requires the fst and disk.frame packages. Coerces objects to disk.frames. Note: disk.frame objects get moved to the drake cache (a subfolder of .drake/ for most workflows). To ensure this data transfer is fast, it is best to save your disk.frame objects to the same physical storage drive as the drake cache, as.disk.frame(your_dataset, outdir = drake_tempfile()).
- "keras": save Keras models as HDF5 files. Requires the keras package.
- "qs": save any R object that can be properly serialized with the qs package. Requires the qs package. Uses qsave() and qread(). Uses the default settings in qs version 0.20.2.
- "rds": save any R object that can be properly serialized. Requires R version >= 3.5.0 due to ALTREP. Note: the "rds" format uses gzip compression, which is slow. "qs" is a superior format.

See Also

drake_plan(), make()

Examples

```
# Use target() to create your own custom columns in a drake plan.
# See ?triggers for more on triggers.
drake_plan(
  website_data = target(
    download_data("www.your_url.com"),
    trigger = "always",
    custom_column = 5
  ),
  analysis = analyze(website_data)
)
models <- c("glm", "hierarchical")
plan <- drake_plan(
  data = target(
    get_data(x),
    transform = map(x = c("simulated", "survey"))
  ),
  analysis = target(
    analyze_data(data, model),
    transform = cross(data, model = !!models, .id = c(x, model))
  ),
  summary = target(
    summarize_analysis(analysis),
    transform = map(analysis, .id = c(x, model))
  ),
  results = target(
    bind_rows(summary),
    transform = combine(summary, .by = data)
  )
)
plan
if (requireNamespace("styler", quietly = TRUE)) {
  print(drake_plan_source(plan))
}
```

---

text_drake_graph *Show a workflow graph as text in your terminal window.* **Stable**

Description

This is a low-tech version of vis_drake_graph() and friends. It is designed for when you do not have access to the usual graphics devices for viewing visuals in an interactive R session: for example, if you are logged into a remote machine with SSH and you do not have access to X Window support.

Usage

```
text_drake_graph(
  ...,
  from = NULL,
  mode = c("out", "in", "all"),
  order = NULL,
  subset = NULL,
  targets_only = FALSE,
  make_imports = TRUE,
  from_scratch = FALSE,
  group = NULL,
  clusters = NULL,
  show_output_files = TRUE,
  nchar = 1L,
  print = TRUE,
  config = NULL
)
```

Arguments

`...`	Arguments to `make()`, such as `plan` and `targets`.
`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.
`targets_only`	Logical, whether to skip the imports and only include the targets in the workflow plan.
`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 original `drake` 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.

nchar — For each node, maximum number of characters of the node label to show. Can be 0, in which case each node is a colored box instead of a node label. Caution: nchar > 0 will mess with the layout.

print — Logical. If TRUE, the graph will print to the console via message(). If FALSE, nothing is printed. However, you still have the visualization because text_drake_graph() and render_text_drake_graph() still invisibly return a character string that you can print yourself with message().

config — Deprecated.

Value

A visNetwork graph.

See Also

render_text_drake_graph(), vis_drake_graph(), sankey_drake_graph(), drake_ggraph()

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
# Plot the network graph representation of the workflow.
pkg <- requireNamespace("txtplot", quietly = TRUE) &&
  requireNamespace("visNetwork", quietly = TRUE)
if (pkg) {
text_drake_graph(my_plan)
make(my_plan) # Run the project, build the targets.
text_drake_graph(my_plan) # The black nodes from before are now green.
}
}
})

## End(Not run)
```

---

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

Description

List all the spec 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:
isolate_example("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)
```

---

transformations *Transformations in* drake_plan(). **Stable**

Description

In drake_plan(), you can define whole batches of targets with transformations such as map(), split(), cross(), and combine().

Arguments

...	Grouping variables. New grouping variables must be supplied with their names and values, existing grouping variables can be given as symbols without any values assigned. For dynamic branching, the entries in ... must be unnamed symbols with no values supplied, and they must be the names of targets.
.data	A data frame of new grouping variables with grouping variable names as column names and values as elements.
.names	Literal character vector of names for the targets. Must be the same length as the targets generated.
.id	Symbol or vector of symbols naming grouping variables to incorporate into target names. Useful for creating short target names. Set .id = FALSE to use integer indices as target name suffixes.

`.tag_in`	A symbol or vector of symbols. Tags assign targets to grouping variables. Use `.tag_in` to assign *untransformed* targets to grouping variables.
`.tag_out`	Just like `.tag_in`, except that `.tag_out` assigns *transformed* targets to grouping variables.
`slice`	Number of slices into which `split()` partitions the data.
`margin`	Which margin to take the slices in `split()`. Same meaning as the `MARGIN` argument of `apply()`.
`drop`	Logical, whether to drop a dimension if its length is 1. Same meaning as `mtcars[,1L,drop = TRUE]` versus `mtcars[,1L,drop = TRUE]`.
`.by`	Symbol or vector of symbols of grouping variables. `combine()` aggregates/groups targets by the grouping variables in `.by`. For dynamic branching, `.by` can only take one variable at a time, and that variable must be a vector. Ideally, it should take little space in memory.
`.trace`	Symbol or vector of symbols for the dynamic trace. The dynamic trace allows you to keep track of the values of dynamic dependencies are associated with individual sub-targets. For `combine()`, `.trace` must either be empty or the same as the variable given for `.by`. See `get_trace()` and `read_trace()` for examples and other details.

Details

For details, see https://books.ropensci.org/drake/plans.html#large-plans.

Transformations

drake has special syntax for generating large plans. Your code will look something like drake_plan(y = target(f(x), transform = map(x = c(1, 2, 3))) You can read about this interface at https://books.ropensci.org/drake/plans.html#large-plans. # nolint

Static branching

In static branching, you define batches of targets based on information you know in advance. Overall usage looks like drake_plan(<x> = target(<...>, transform = <call>), where

- <x> is the name of the target or group of targets.
- <...> is optional arguments to `target()`.
- <call> is a call to one of the transformation functions.

Transformation function usage:

- `map(...,.data,.names,.id,.tag_in,.tag_out)`
- `split(...,slices,margin = 1L,drop = FALSE,.names,.tag_in,.tag_out)` # nolint
- `cross(...,.data,.names,.id,.tag_in,.tag_out)`
- `combine(...,.by,.names,.id,.tag_in,.tag_out)`

Dynamic branching

- `map(...,.trace)`
- `cross(...,.trace)`
- `group(...,.by,.trace)`

`map()` and `cross()` create dynamic sub-targets from the variables supplied to the dots. As with static branching, the variables supplied to `map()` must all have equal length. `group(f(data),.by = x)` makes new dynamic sub-targets from `data`. Here, `data` can be either static or dynamic. If `data` is dynamic, `group()` aggregates existing sub-targets. If `data` is static, `group()` splits `data` into multiple subsets based on the groupings from `.by`.

Differences from static branching:

- `...` must contain *unnamed* symbols with no values supplied, and they must be the names of targets.
- Arguments `.id`, `.tag_in`, and `.tag_out` no longer apply.

Examples

```
# Static branching
models <- c("glm", "hierarchical")
plan <- drake_plan(
  data = target(
    get_data(x),
    transform = map(x = c("simulated", "survey"))
  ),
  analysis = target(
    analyze_data(data, model),
    transform = cross(data, model = !!models, .id = c(x, model))
  ),
  summary = target(
    summarize_analysis(analysis),
    transform = map(analysis, .id = c(x, model))
  ),
  results = target(
    bind_rows(summary),
    transform = combine(summary, .by = data)
  )
)
plan
if (requireNamespace("styler")) {
  print(drake_plan_source(plan))
}
# Static splitting
plan <- drake_plan(
  analysis = target(
    analyze(data),
    transform = split(data, slices = 3L, margin = 1L, drop = FALSE)
  )
)
print(plan)
if (requireNamespace("styler", quietly = TRUE)) {
```

```
  print(drake_plan_source(plan))
}
# Static tags:
drake_plan(
  x = target(
    command,
    transform = map(y = c(1, 2), .tag_in = from, .tag_out = c(to, out))
  ),
  trace = TRUE
)
plan <- drake_plan(
  survey = target(
    survey_data(x),
    transform = map(x = c(1, 2), .tag_in = source, .tag_out = dataset)
  ),
  download = target(
    download_data(),
    transform = map(y = c(5, 6), .tag_in = source, .tag_out = dataset)
  ),
  analysis = target(
    analyze(dataset),
    transform = map(dataset)
  ),
  results = target(
    bind_rows(analysis),
    transform = combine(analysis, .by = source)
  )
)
plan
if (requireNamespace("styler", quietly = TRUE)) {
  print(drake_plan_source(plan))
}
```

`transform_plan` *Transform a plan* **Stable**

Description

Evaluate the `map()`, `cross()`, `split()` and `combine()` operations in the `transform` column of a `drake` plan.

Usage

```
transform_plan(
  plan,
  envir = parent.frame(),
  trace = FALSE,
  max_expand = NULL,
  tidy_eval = TRUE
)
```

Arguments

plan	A drake plan with a transform column
envir	Environment for tidy evaluation.
trace	Logical, whether to add columns to show what happens during target transformations.
max_expand	Positive integer, optional. max_expand is the maximum number of targets to generate in each map(), split(), or cross() transform. Useful if you have a massive plan and you want to test and visualize a strategic subset of targets before scaling up. Note: the max_expand argument of drake_plan() and transform_plan() is for static branching only. The dynamic branching max_expand is an argument of make() and drake_config().
tidy_eval	Logical, whether to use tidy evaluation (e.g. unquoting/!!) when resolving commands. Tidy evaluation in transformations is always turned on regardless of the value you supply to this argument.

Details

https://books.ropensci.org/drake/plans.html#large-plans # nolint

See Also

drake_plan, map, split, cross, combine

Examples

```
plan1 <- drake_plan(
  y = target(
    f(x),
    transform = map(x = c(1, 2))
  ),
  transform = FALSE
)
plan2 <- drake_plan(
  z = target(
    g(y),
    transform = map(y, .id = x)
  ),
  transform = FALSE
)
plan <- bind_plans(plan1, plan2)
transform_plan(plan)
models <- c("glm", "hierarchical")
plan <- drake_plan(
  data = target(
    get_data(x),
    transform = map(x = c("simulated", "survey"))
  ),
  analysis = target(
    analyze_data(data, model),
```

```
    transform = cross(data, model = !!models, .id = c(x, model))
  ),
  summary = target(
    summarize_analysis(analysis),
    transform = map(analysis, .id = c(x, model))
  ),
  results = target(
    bind_rows(summary),
    transform = combine(summary, .by = data)
  )
)
plan
if (requireNamespace("styler", quietly = TRUE)) {
  print(drake_plan_source(plan))
}
# Tags:
drake_plan(
  x = target(
    command,
    transform = map(y = c(1, 2), .tag_in = from, .tag_out = c(to, out))
  ),
  trace = TRUE
)
plan <- drake_plan(
  survey = target(
    survey_data(x),
    transform = map(x = c(1, 2), .tag_in = source, .tag_out = dataset)
  ),
  download = target(
    download_data(),
    transform = map(y = c(5, 6), .tag_in = source, .tag_out = dataset)
  ),
  analysis = target(
    analyze(dataset),
    transform = map(dataset)
  ),
  results = target(
    bind_rows(analysis),
    transform = combine(analysis, .by = source)
  )
)
plan
if (requireNamespace("styler", quietly = TRUE)) {
  print(drake_plan_source(plan))
}
```

`trigger` *Customize the decision rules for rebuilding targets* **Stable**

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://books.ropensci.org/drake/triggers.html

Usage

```
trigger(
  command = TRUE,
  depend = TRUE,
  file = TRUE,
  seed = TRUE,
  format = 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. Also applies to external data tracked with target(format = "file").
seed	Logical, whether to rebuild the target if the seed changes. Only makes a difference if you set a custom seed column in your drake_plan() at some point in your workflow.
format	Logical, whether to rebuild the target if the choice of specialized data format changes: for example, if you use target(format = "qs") one instance and target(format = "fst") the next. See https://books.ropensci.org/drake/plans.html#special-data-formats-for-targets # nolint for details on formats.
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:
isolate_example("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)
```

`use_drake` *Use drake in a project* **Questioning**

Description

Add top-level R script files to use drake in your data analysis project. For details, read https://books.ropensci.org/drake/projects.html

Usage

```
use_drake(open = interactive())
```

Arguments

open	Logical, whether to open make.R for editing.

Details

Files written:

1. make.R: a suggested main R script for batch mode.
2. _drake.R: a configuration R script for the r_*() functions documented at # nolint https://books.ropensci.org/drake/projects.html#safer-interactivity. # nolint Remarks:

- There is nothing magical about the name, make.R. You can call it whatever you want.
- Other supporting scripts, such as R/packages.R, R/functions.R, and R/plan.R, are not included.
- You can find examples at https://github.com/wlandau/drake-examples and download examples with drake_example() (e.g. drake_example("main")).

Examples

```
## Not run:
# use_drake(open = FALSE) # nolint

## End(Not run)
```

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

Description

It is good practice to visualize the dependency graph before running the targets.

Usage

```
vis_drake_graph(
  ...,
  file = character(0),
  selfcontained = FALSE,
  build_times = "build",
  digits = 3,
  targets_only = FALSE,
  font_size = 20,
  layout = NULL,
  main = NULL,
  direction = NULL,
  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,
  on_select_col = NULL,
  on_select = NULL,
  level_separation = NULL,
  config = NULL
)
```

Arguments

...	Arguments to make(), such as plan and targets.
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_phantomjs(). 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` Deprecated.

`main` Character string, title of the graph.

`direction` Deprecated.

`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 = TRUE)`

`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 original drake 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).

on_select_col — Optional string corresponding to the column name in the plan that should provide data for the on_select event.

on_select — defines node selection event handling. Either a string of valid JavaScript that may be passed to visNetwork::visEvents(), or one of the following: TRUE, NULL/FALSE. If TRUE , enables the default behavior of opening the link specified by the on_select_col given to drake_graph_info(). NULL/FALSE disables the behavior.

level_separation — Numeric, levelSeparation argument to visNetwork::visHierarchicalLayout(). Controls the distance between hierarchical levels. Consider setting if the aspect ratio of the graph is far from 1. Defaults to 150 through visNetwork.

config — Deprecated.

Details

For enhanced interactivity in the graph, see the mandrake package: https://github.com/matthewstrasiotto/mandrake.

Value

A visNetwork graph.

See Also

render_drake_graph(), sankey_drake_graph(), drake_ggraph(), text_drake_graph()

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
if (suppressWarnings(require("knitr"))) {
load_mtcars_example() # Get the code with drake_example("mtcars").
# Plot the network graph representation of the workflow.
if (requireNamespace("visNetwork", quietly = TRUE)) {
vis_drake_graph(my_plan)
make(my_plan) # Run the project, build the targets.
vis_drake_graph(my_plan) # The red nodes from before are now green.
# Plot a subgraph of the workflow.
vis_drake_graph(
  my_plan,
  from = c("small", "reg2")
)
}
}
})

## End(Not run)
```

`which_clean` *Which targets will* `clean()` *invalidate?* **Stable**

Description

`which_clean()` is a safety check for `clean()`. It shows you the targets that `clean()` will invalidate (or remove if `garbage_collection` is `TRUE`). It helps you avoid accidentally removing targets you care about.

Usage

```
which_clean(
  ...,
  list = character(0),
  path = NULL,
  cache = drake::drake_cache(path = path)
)
```

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()`.
`path`	Path to a `drake` cache (usually a hidden .drake/ folder) or `NULL`.
`cache`	drake cache. See `new_cache()`. If supplied, `path` is ignored.

See Also

clean()

Examples

```
## Not run:
isolate_example("Quarantine side effects.", {
plan <- drake_plan(x = 1, y = 2, z = 3)
make(plan)
cached()
which_clean(x, y) # [1] "x" "y"
clean(x, y)       # Invalidates targets x and y.
cached()          # [1] "z"
})

## End(Not run)
```

Index