Package 'targets'

March 28, 2021

Title Dynamic Function-Oriented 'Make'-Like Declarative Workflows

Description As a pipeline toolkit for Statistics and data science in R, the 'targets' package brings together function-oriented programming and 'Make'-like declarative workflows. It analyzes the dependency relationships among the tasks of a workflow, skips steps that are already up to date, runs the necessary computation with optional parallel workers, abstracts files as R objects, and provides tangible evidence that the results match the underlying code and data. The methodology in this package borrows from GNU 'Make' (2015, ISBN:978-9881443519) and 'drake' (2018, <doi:10.21105/joss.00550>).

Version 0.3.1

License MIT + file LICENSE

URL https://docs.ropensci.org/targets/, https://github.com/ropensci/targets

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

Depends R (>= 3.5.0)

Imports callr (>= 3.4.3), cli (>= 2.0.2), codetools (>= 0.2.16), data.table (>= 1.12.8), digest (>= 0.6.25), igraph (>= 1.2.5), R6 (>= 2.4.1), rlang (>= 0.4.10), stats, tibble (>= 3.0.1), tidyselect (>= 1.1.0), utils, vctrs (>= 0.2.4), withr (>= 2.1.2)

Suggests arrow (>= 3.0.0), aws.s3 (>= 0.3.21), bs4Dash(>= 0.5.0), clustermq (>= 0.8.95.1), curl (>= 4.3), dplyr (>= 1.0.0), fst (>= 0.9.2), future (>= 1.19.1), future.callr (>= 0.6.0), gt (>= 0.2.2), keras (>= 2.2.5.0), knitr (>= 1.30), rmarkdown (>= 2.4), pingr (>= 2.0.1), pkgload (>= 1.1.0), qs (>= 0.23.2), rstudioapi (>= 0.11), shiny (>= 1.5.0), shinycssloaders (>= 1.0.0), shinyWidgets (>= 0.5.4), testthat (>= 3.0.0), torch (>= 0.1.0), usethis (>= 1.6.3), visNetwork (>= 2.0.9)

Encoding UTF-8

Language en-US

VignetteBuilder knitr

Config/testthat/edition 3

RoxygenNote 7.1.1

NeedsCompilation no

Author William Michael Landau [aut, cre]
(<https://orcid.org/0000-0003-1878-3253>),
Matthew T. Warkentin [ctb],
Samantha Oliver [rev] (<https://orcid.org/0000-0001-5668-1165>),
Tristan Mahr [rev] (<https://orcid.org/0000-0002-8890-5116>),
Eli Lilly and Company [cph]

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

Repository CRAN

Date/Publication 2021-03-28 06:50:02 UTC

R topics documented:

`targets-package` *targets: Dynamic Function-Oriented Make-Like Declarative Pipelines for R*

Description

As a pipeline toolkit for Statistics and data science in R, the `targets` package brings together function-oriented programming and Make-like declarative pipelines. It analyzes the dependency relationships among the tasks of a workflow, skips steps that are already up to date, runs the necessary computations with optional parallel workers, abstracts files as R objects, and provides tangible evidence that the results match the underlying code and data. The methodology in this package borrows from GNU Make (2015, ISBN:978-9881443519) and `drake` by Will Landau (2018, doi: 10.21105/joss.00550).

tar_branches *Reconstruct the branch names and the names of their dependencies.*

Description

Given a branching pattern, use available metadata to reconstruct branch names and the names of each branch's dependencies. The metadata of each target must already exist and be consistent with the metadata of the other targets involved.

Usage

```
tar_branches(name, pattern)
```

Arguments

name	Symbol, name of the target.
pattern	Language to define branching for a target. For example, in a pipeline with numeric vector targets x and y, tar_target(z,x + y,pattern = map(x,y)) implicitly defines branches of z that each compute x[1] + y[1], x[2] + y[2], and so on. See the user manual for details.

Details

The results from this function can help you retroactively figure out correspondences between upstream branches and downstream branches. However, it does not always correctly predict what the names of the branches will be after the next run of the pipeline. Dynamic branching happens while the pipeline is running, so we cannot always know what the names of the branches will be in advance (or even how many there will be).

Value

A tibble with one row per branch and one column for each target (including the branched-over targets and the target with the pattern.)

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, head(letters, 2)),
    tar_target(z, head(LETTERS, 2)),
    tar_target(dynamic, c(x, y, z), pattern = cross(z, map(x, y)))
  )
}, ask = FALSE)
tar_make()
tar_branches(dynamic, pattern = cross(z, map(x, y)))
```

```
})
}
```

`tar_cancel` *Cancel a target mid-build under a custom condition.*

Description

Cancel a target while its command is running if a condition is met.

Usage

```
tar_cancel(condition = TRUE)
```

Arguments

`condition`	Logical of length 1, whether to cancel the target.

Details

Must be invoked by the target itself. `tar_cancel()` cannot interrupt a target from another process.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(tar_target(x, tar_cancel(1 > 0)))
tar_make() # Should cancel target x.
})
}
```

`tar_cue` *Declare the rules that cue a target.*

Description

Declare the rules that mark a target as outdated.

Usage

```
tar_cue(
  mode = c("thorough", "always", "never"),
  command = TRUE,
  depend = TRUE,
  format = TRUE,
  iteration = TRUE,
  file = TRUE
)
```

Arguments

mode	Cue mode. If "thorough", all the cues apply unless individually suppressed. If "always", then the target always runs. If "never", then the target does not run unless the metadata does not exist or the last run errored.
command	Logical, whether to rerun the target if command changed since last time.
depend	Logical, whether to rerun the target if the value of one of the dependencies changed.
format	Logical, whether to rerun the target if the user-specified storage format changed. The storage format is user-specified through tar_target() or tar_option_set().
iteration	Logical, whether to rerun the target if the user-specified iteration method changed. The iteration method is user-specified through tar_target() or tar_option_set().
file	Logical, whether to rerun the target if the file(s) with the return value changed or at least one is missing.

Details

targets uses internal metadata and special cues to decide if a target is up to date. A target is outdated if one of the following cues is met (checked in the order given below). tar_cue() can activate or suppress many of these cues. See the user manual for details.

1. There is no metadata record of the target.
2. The target errored last run.
3. The target has a different class than it did before.
4. The cue mode equals "always".
5. The cue mode does not equal "never".
6. The command metadata field (the hash of the R command) is different from last time.
7. The depend metadata field (the hash of the immediate upstream dependency targets and global objects) is different from last time.
8. The storage format is different from last time.
9. The iteration mode is different from last time.
10. A target's file (either the one in _targets/objects/ or a dynamic file) does not exist or changed since last time.

A target's dependencies can include functions, and these functions are tracked for changes using a custom hashing procedure. When a function's hash changes, the function is considered invalidated, and so are any downstream targets with the depend cue turned on. The targets package computes the hash of a function in the following way.

1. Deparse the function with targets:::deparse_safe(). This function computes a string representation of the function that removes comments and standardizes whitespace so that trivial changes to formatting do not cue targets to rerun.
2. Manually remove any literal pointers from the function string using targets:::mask_pointers(). Such pointers arise from inline compiled C/C++ functions.
3. Compute a hash on the preprocessed string above using targets:::digest_chr64().

Those functions themselves have dependencies, and those dependencies are detected with `codetools::findGlobals()`. Dependencies of functions may include other global functions or global objects. If a dependency of a function is invalidated, the function itself is invalidated, and so are any dependent targets with the depend cue turned on.

Examples

```
# The following target will always run when the pipeline runs.
x <- tar_target(x, download_data(), cue = tar_cue(mode = "always"))
```

`tar_delete` *Delete target return values.*

Description

Delete the return values of targets in _targets/objects/. but keep the records in _targets/meta/meta. Dynamic files outside the data store are unaffected. The _targets/ data store must be in the current working directory.

Usage

```
tar_delete(names)
```

Arguments

`names`	Names of the targets to remove from _targets/objects/. You can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.

Details

For patterns recorded in the metadata, all the branches will be deleted. For patterns no longer in the metadata, branches are left alone.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
tar_delete(starts_with("y")) # Only deletes y1 and y2.
tar_make() # y1 and y2 rebuild but return same values, so z is up to date.
})
}
```

tar_deps *Code dependencies*

Description

List the dependencies of a function or expression.

Usage

```
tar_deps(expr)
```

Arguments

expr	A quoted R expression or function.

Details

`targets` detects the dependencies of commands using static code analysis. Use `tar_deps()` to run the code analysis and see the dependencies for yourself.

Value

Character vector of the dependencies of a function or expression.

Examples

```
tar_deps(x <- y + z)
tar_deps({
  x <- 1
  x + a
})
tar_deps(function(a = b) map_dfr(data, ~do_row(.x)))
```

tar_deps_raw *Code dependencies (raw version)*

Description

Same as `tar_deps()` except `expr` must already be an unquoted function or expression object.

Usage

```
tar_deps_raw(expr)
```

Arguments

expr	An R expression object or function.

Value

Character vector of the dependencies of a function or expression.

Examples

```
tar_deps_raw(quote(x <- y + z))
tar_deps_raw(
  quote({
    x <- 1
    x + a
  })
)
tar_deps_raw(function(a = b) map_dfr(data, ~do_row(.x)))
```

`tar_destroy` *Destroy a section or all of the* _targets/ *data store in the current working directory*

Description

Destroy _targets/ data store in the current working directory. Optionally, just destroy part of the data store.

Usage

```
tar_destroy(
 destroy = c("all", "meta", "process", "progress", "objects", "scratch", "workspaces")
)
```

Arguments

`destroy` Character of length 1, what to destroy. Choices:

- `"all"`: destroy the entire data store.
- `"meta"`: just delete the metadata file at _targets/meta/meta, which invalidates all the targets but keeps the data.
- `"process"`: just delete the progress data file at _targets/meta/process, which resets the metadata of the main process.
- `"progress"`: just delete the progress data file at _targets/meta/progress, which resets the progress tracking info.
- `"objects"`: delete all the target return values in _targets/objects/ but keep progress and metadata. Dynamic files are not deleted this way.
- `"scratch"`: temporary files saved during `tar_make()` that should automatically get deleted except if R crashed.
- `"workspaces"`: compressed files in _targets/workspaces/ with the saved workspaces of targets that errored. Only saved if `error = "workspace"` in `tar_option_set()` or `tar_target()`. Load a workspace with `tar_workspace()`.

Value

Nothing.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(list(tar_target(x, 1 + 1)), ask = FALSE)
tar_make() # Creates the _targets/ data store.
tar_destroy()
print(file.exists("_targets")) # Should be FALSE.
})
}
```

---

tar_dir *Execute code in a temporary directory.*

Description

Runs code inside a new `tempfile()` directory in order to avoid writing to the user's file space. Used in examples and tests in order to comply with CRAN policies.

Usage

```
tar_dir(code)
```

Arguments

code	User-defined code.

Value

Return value of the user-defined code.

Examples

```
tar_dir(file.create("only_exists_in_tar_dir"))
file.exists("only_exists_in_tar_dir")
```

---

tar_edit *Open _targets.R for editing.*

Description

Looks for _targets.R in the current working directory. Requires the `usethis` package.

Usage

```
tar_edit()
```

`tar_envir` *For developers only: get the environment of the current target.*

Description

For developers only: get the environment where a target runs its command. Inherits from `tar_option_get("envir")`.

Usage

```
tar_envir(default = parent.frame())
```

Arguments

`default`	Environment, value to return if `tar_envir()` is called on its own outside a `targets` pipeline. Having a default lets users run things without `tar_make()`, which helps peel back layers of code and troubleshoot bugs.

Details

Users should not call `tar_envir()` directly because accidental modifications to `parent.env(tar_envir())` could break the pipeline. `tar_envir()` only exists in order to support third-party interface packages such as `tarchetypes`.

Value

If called from a running target, `tar_envir()` returns the environment where the target runs its command. If called outside a pipeline, the return value is whatever the user supplies to `default` (which defaults to `parent.frame()`).

Examples

```
tar_envir()
tar_envir(default = new.env(parent = emptyenv()))
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(tar_target(x, tar_envir(default = parent.frame())))
tar_make(x)
tar_read(x)
})
}
```

tar_exist_meta *Check if target metadata exists.*

Description

Check if the target metadata file _targets/meta/meta exists for the current project.

Usage

```
tar_exist_meta()
```

Details

To learn more about local storage in `targets`, visit https://books.ropensci.org/targets/files.html#internal-files.

Value

Logical of length 1, whether the current project's metadata exists.

Examples

```
tar_exist_meta()
```

tar_exist_objects *Check if local output data exists for one or more targets.*

Description

Check if the local data files exist in _targets/objects/ for one or more targets.

Usage

```
tar_exist_objects(names)
```

Arguments

`names`	Character vector of target names.

Details

To learn more about local storage in `targets`, visit https://books.ropensci.org/targets/files.html#internal-files.

Value

Logical of length `length(names)`, whether each given target has an existing file in _targets/objects/ for the current project.

Examples

```
tar_exist_objects(c("target1", "target2"))
```

---

tar_exist_process *Check if process metadata exists.*

Description

Check if the process metadata file _targets/meta/process exists for the current project.

Usage

```
tar_exist_process()
```

Details

To learn more about local storage in `targets`, visit https://books.ropensci.org/targets/files.html#internal-files.

Value

Logical of length 1, whether the current project's metadata exists.

Examples

```
tar_exist_process()
```

---

tar_exist_progress *Check if progress metadata exists.*

Description

Check if the progress metadata file _targets/meta/progress exists for the current project.

Usage

```
tar_exist_progress()
```

Details

To learn more about local storage in `targets`, visit https://books.ropensci.org/targets/files.html#internal-files.

Value

Logical of length 1, whether the current project's metadata exists.

Examples

```
tar_exist_progress()
```

`tar_exist_script` *Check if the target script exists.*

Description

Check if the _targets.R file of the current project exists. exists for the current project.

Usage

```
tar_exist_script()
```

Value

Logical of length 1, whether the current project's metadata exists.

Examples

```
tar_exist_script()
```

`tar_github_actions` *Set up GitHub Actions to run a targets pipeline*

Description

Writes a GitHub Actions workflow file so the pipeline runs on every push to GitHub. Historical runs accumulate in the `targets-runs` branch, and the latest output is restored before `tar_make()` so up-to-date targets do not rerun.

Usage

```
tar_github_actions(
  path = file.path(".github", "workflows", "targets.yaml"),
  ask = NULL
)
```

Arguments

path	Character of length 1, file path to write the GitHub Actions workflow file.
ask	Logical, whether to ask before writing if the workflow file already exists. If NULL, defaults to Sys.getenv("TAR_ASK"). (Set to "true" or "false" with Sys.setenv()). If ask and the TAR_ASK environment variable are both indeterminate, defaults to interactive().

Details

Steps to set up continuous deployment:

1. Ensure your pipeline stays within the resource limitations of GitHub Actions and repositories, both for storage and compute. For storage, you may wish to reduce the burden with AWS-backed storage formats like "aws_qs".
2. Ensure Actions are enabled in your GitHub repository. You may have to visit the Settings tab.
3. Call targets::tar_renv(extras = character(0)) to expose hidden package dependencies.
4. Set up renv for your project (with renv::init() or renv::snapshot()). Details at https://rstudio.github.io/renv/articles/ci.html.
5. Commit the renv.lock file to the main (recommended) or master Git branch.
6. Run tar_github_actions() to create the workflow file. Commit this file to main (recommended) or master in Git.
7. Push your project to GitHub. Verify that a GitHub Actions workflow runs and pushes results to targets-runs. Subsequent runs will only recompute the outdated targets.

Value

Nothing (invisibly). This function writes a GitHub Actions workflow file as a side effect.

Examples

```
tar_github_actions(tempfile())
```

---

tar_glimpse *Visualize an abridged fast dependency graph.*

Description

Analyze the pipeline defined in _targets.R and visualize the directed acyclic graph of targets. Unlike tar_visnetwork(), tar_glimpse() does not account for metadata or progress information, which means the graph renders faster. Also, tar_glimpse() omits functions and other global objects by default (but you can include them with targets_only = FALSE).

Usage

```
tar_glimpse(
  targets_only = TRUE,
  allow = NULL,
  exclude = ".Random.seed",
  level_separation = NULL,
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function)
)
```

Arguments

`targets_only`	Logical, whether to restrict the output to just targets (`FALSE`) or to also include imported global functions and objects.
`allow`	Optional, define the set of allowable vertices in the graph. Set to `NULL` to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
`exclude`	Optional, define the set of exclude vertices from the graph. Set to `NULL` to exclude no vertices. Otherwise, you can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
`level_separation`	Numeric of length 1, `levelSeparation` argument of `visNetwork::visHierarchicalLayout()`. Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If `level_separation` is `NULL`, the `levelSeparation` argument of `visHierarchicalLayout()` defaults to `150`.
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.

Value

A `visNetwork` HTML widget object.

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
```

```
  )
}, ask = FALSE)
tar_glimpse()
tar_glimpse(allow = starts_with("y"))
})
}
```

`tar_group` *Group a data frame to iterate over subsets of rows.*

Description

Like `dplyr::group_by()`, but for patterns. `tar_group()` allows you to map or cross over subsets of data frames. Requires `iteration = "group"` on the target. See the example.

Usage

```
tar_group(x)
```

Arguments

`x`	Grouped data frame from `dplyr::group_by()`

Details

The goal of `tar_group()` is to post-process the return value of a data frame target to allow downstream targets to branch over subsets of rows. It takes the groups defined by `dplyr::group_by()` and translates that information into a special `tar_group` is a column. `tar_group` is a vector of positive integers from 1 to the number of groups. Rows with the same integer in `tar_group` belong to the same group, and branches are arranged in increasing order with respect to the integers in `tar_group`. The assignment of `tar_group` integers to group levels depends on the orderings inside the grouping variables and not the order of rows in the dataset. `dplyr::group_keys()` on the grouped data frame shows how the grouping variables correspond to the integers in the `tar_group` column.

Value

A data frame with a special `tar_group` column that `targets` will use to find subsets of your data frame.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
# The tar_group() function simply creates
# a tar_group column to partition the rows
# of a data frame.
data.frame(
  x = seq_len(6),
  id = rep(letters[seq_len(3)], each = 2)
```

```
) %>%
  dplyr::group_by(id) %>%
  tar_group()
# We use tar_group() below to branch over
# subsets of a data frame defined with dplyr::group_by().
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
library(dplyr)
list(
  tar_target(
    data,
    data.frame(
      x = seq_len(6),
      id = rep(letters[seq_len(3)], each = 2)
    ) %>%
      group_by(id) %>%
      tar_group(),
    iteration = "group"
  ),
  tar_target(
    sums,
    sum(data$x),
    pattern = map(data),
    iteration = "vector"
  )
)
})
tar_make()
tar_read(sums) # Should be c(3, 7, 11).
})
}
```

`tar_helper` *Write a helper R script.*

Description

Write a helper R script for a `targets` pipeline. Could be supporting functions or the _targets.R file itself.

Usage

```
tar_helper(path = NULL, code = NULL, tidy_eval = TRUE, envir = parent.frame())
```

Arguments

`path`	Character of length 1, path to write (or overwrite) code. If the parent directory does not exist, `tar_helper_raw()` creates it.
`code`	Quoted code to write to `path`. `tar_helper()` overwrites the file if it already exists.

tidy_eval	Logical, whether to use tidy evaluation on code. If turned on, you can substitute expressions and symbols using !! and !!!. See examples below.
envir	Environment for tidy evaluation.

Details

tar_helper() is a specialized version of tar_script() with flexible paths and tidy evaluation.

Value

NULL (invisibly)

Examples

```
# Without tidy evaluation:
path <- tempfile()
tar_helper(path, x <- 1)
writeLines(readLines(path))
# With tidy evaluation:
y <- 123
tar_helper(path, x <- !!y)
writeLines(readLines(path))
```

---

tar_helper_raw *Write a helper R script (raw version).*

Description

Write a helper R script for a targets pipeline. Could be supporting functions or the _targets.R file itself.

Usage

```
tar_helper_raw(path = NULL, code = NULL)
```

Arguments

path	Character of length 1, path to write (or overwrite) code. If the parent directory does not exist, tar_helper_raw() creates it.
code	Expression object. tar_helper_raw() deparses and writes this code to a file at path, overwriting it if the file already exists.

Details

tar_helper_raw() is a specialized version of tar_script() with flexible paths and tidy evaluation. It is like tar_helper() except that code is an "evaluated" argument rather than a quoted one.

Value

NULL (invisibly)

Examples

```
path <- tempfile()
tar_helper_raw(path, quote(x <- 1))
writeLines(readLines(path))
```

`tar_invalidate` *Invalidate targets and global objects in the metadata.*

Description

Delete the metadata of records in _targets/meta/meta but keep the return values of targets in _targets/objects/. The _targets/ data store must be in the current working directory.

Usage

```
tar_invalidate(names)
```

Arguments

names	Names of the targets to remove from the metadata list. You can supply symbols, a character vector, or tidyselect helpers like starts_with().

Details

For patterns recorded in the metadata, all the branches will be invalidated. For patterns no longer in the metadata, branches are left alone.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
tar_invalidate(starts_with("y")) # Only invalidates y1 and y2.
tar_make() # y1 and y2 rebuild but return same values, so z is up to date.
})
}
```

tar_load *Load the values of targets.*

Description

Load the return values of targets into the current environment (or the environment of your choosing). For a typical target, the return value lives in a file in _targets/objects/. For dynamic files (i.e. `format = "file"`) the paths loaded in place of the values.

Usage

```
tar_load(
  names,
  branches = NULL,
  meta = tar_meta(targets_only = TRUE),
  envir = parent.frame()
)
```

Arguments

names	Names of the targets to load. You can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
branches	Integer of indices of the branches to load for any targets that are patterns.
meta	Data frame of metadata from `tar_meta()`. `tar_read()` with the default arguments can be inefficient for large pipelines because all the metadata is stored in a single file. However, if you call `tar_meta()` beforehand and supply it to the `meta` argument, then successive calls to `tar_read()` may run much faster.
envir	Environment to put the loaded targets.

Value

Nothing.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
tar_load(starts_with("y"))
})
}
```

tar_load_raw *Load the values of targets (raw version).*

Description

Same as tar_load() except names is a character vector. Do not use in knitr or R Markdown reports with tarchetypes::tar_knit() or tarchetypes::tar_render().

Usage

```
tar_load_raw(names, branches = NULL, meta = tar_meta(), envir = parent.frame())
```

Arguments

names	Character vector, names of the targets to build or check.
branches	Integer of indices of the branches to load for any targets that are patterns.
meta	Data frame of metadata from tar_meta(). tar_read() with the default arguments can be inefficient for large pipelines because all the metadata is stored in a single file. However, if you call tar_meta() beforehand and supply it to the meta argument, then successive calls to tar_read() may run much faster.
envir	Environment to put the loaded targets.

Value

Nothing.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
tar_load_raw(c("y1", "y2"))
y1
y2
})
}
```

`tar_make` *Run a pipeline of targets.*

Description

Run the pipeline you defined in _targets.R. `tar_make()` runs the correct targets in the correct order and stores the return values in _targets/objects/.

Usage

```
tar_make(
  names = NULL,
  reporter = Sys.getenv("TAR_MAKE_REPORTER", unset = "verbose"),
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter)
)
```

Arguments

`names`	Names of the targets to build or check. Set to `NULL` to check/build all the targets (default). Otherwise, you can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
`reporter`	Character of length 1, name of the reporter to user. Controls how messages are printed as targets run in the pipeline. Defaults to the `TAR_MAKE_REPORTER` environment variable if set and `"verbose"` otherwise. Choices: • `"verbose"`: print one message for each target that runs (default). • `"silent"`: print nothing. • `"timestamp"`: print a time-stamped message for each target that runs. • `"summary"`: print a running total of the number of each targets in each status category (queued, running, skipped, build, canceled, or errored).
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.

Value

`NULL` except if `callr_function = callr::r_bg()`, in which case a handle to the `callr` background process is returned. Either way, the value is invisibly returned.

Examples

```
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set()
  list(tar_target(x, 1 + 1))
})
tar_make()
tar_script({
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make(starts_with("y")) # Only builds y1 and y2.
})
```

`tar_make_clustermq` *Run a pipeline of targets in parallel with persistent* `clustermq` *workers.*

Description

This function is like `tar_make()` except that targets run in parallel with persistent `clustermq` workers. It requires that you set global options like `clustermq.scheduler` and `clustermq.template` inside the _targets.R script. `clustermq` is not a strict dependency of `targets`, so you must install `clustermq` yourself.

Usage

```
tar_make_clustermq(
  names = NULL,
  reporter = Sys.getenv("TAR_MAKE_REPORTER", unset = "verbose"),
  workers = 1L,
  log_worker = FALSE,
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter)
)
```

Arguments

`names`	Names of the targets to build or check. Set to `NULL` to check/build all the targets (default). Otherwise, you can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
`reporter`	Character of length 1, name of the reporter to user. Controls how messages are printed as targets run in the pipeline. Defaults to the `TAR_MAKE_REPORTER` environment variable if set and `"verbose"` otherwise. Choices:

- "verbose": print one message for each target that runs (default).
- "silent": print nothing.
- "timestamp": print a time-stamped message for each target that runs.
- "summary": print a running total of the number of each targets in each status category (queued, running, skipped, build, canceled, or errored).

workers — Positive integer, number of persistent clustermq workers to create.

log_worker — Logical, whether to write a log file for each worker. Same as the log_worker argument of clustermq::Q() and clustermq::workers().

callr_function — A function from callr to start a fresh clean R process to do the work. Set to NULL to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). callr_function needs to be NULL for interactive debugging, e.g. tar_option_set(debug = "your_target"). However, callr_function should not be NULL for serious reproducible work.

callr_arguments — A list of arguments to callr_function.

Details

To use with a cluster, you will need to set the global options clustermq.scheduler and clustermq.template inside _targets.R. To read more about configuring clustermq for your scheduler, visit https://mschubert.github.io/clustermq/articles/userguide.html#configuration # nolint and navigate to the appropriate link under "Setting up the scheduler". Wildcards in the template file are filled in with elements from tar_option_get("resources").

Value

NULL except if callr_function = callr::r_bg(), in which case a handle to the callr background process is returned. Either way, the value is invisibly returned.

Examples

```
if (!identical(tolower(Sys.info()[["sysname"]]), "windows")) {
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  options(clustermq.scheduler = "multicore") # Does not work on Windows.
  tar_option_set()
  list(tar_target(x, 1 + 1))
}, ask = FALSE)
tar_make_clustermq()
})
}
}
```

tar_make_future *Run a pipeline of targets in parallel with transient* future *workers.*

Description

This function is like tar_make() except that targets run in parallel with transient future workers. It requires that you declare your future::plan() inside the _targets.R script. future is not a strict dependency of targets, so you must install future yourself.

Usage

```
tar_make_future(
  names = NULL,
  reporter = Sys.getenv("TAR_MAKE_REPORTER", unset = "verbose"),
  workers = 1L,
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter)
)
```

Arguments

names	Names of the targets to build or check. Set to NULL to check/build all the targets (default). Otherwise, you can supply symbols, a character vector, or tidyselect helpers like starts_with().
reporter	Character of length 1, name of the reporter to user. Controls how messages are printed as targets run in the pipeline. Defaults to the TAR_MAKE_REPORTER environment variable if set and "verbose" otherwise. Choices: • "verbose": print one message for each target that runs (default). • "silent": print nothing. • "timestamp": print a time-stamped message for each target that runs. • "summary": print a running total of the number of each targets in each status category (queued, running, skipped, build, canceled, or errored).
workers	Positive integer, maximum number of transient future workers allowed to run at any given time.
callr_function	A function from callr to start a fresh clean R process to do the work. Set to NULL to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). callr_function needs to be NULL for interactive debugging, e.g. tar_option_set(debug = "your_target"). However, callr_function should not be NULL for serious reproducible work.
callr_arguments	A list of arguments to callr_function.

Details

To configure tar_make_future() with a computing cluster, see the future.batchtools package documentation.

Value

NULL except if callr_function = callr::r_bg(), in which case a handle to the callr background process is returned. Either way, the value is invisibly returned.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  future::plan(future::multisession)
  tar_option_set()
  list(tar_target(x, 1 + 1))
}, ask = FALSE)
tar_make_future()
})
}
```

---

tar_manifest *Produce a data frame of information about your targets.*

Description

Along with tar_visnetwork() and tar_glimpse(), tar_manifest() helps check that you constructed your pipeline correctly.

Usage

```
tar_manifest(
  names = NULL,
  fields = c("name", "command", "pattern"),
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function)
)
```

Arguments

names	Names of the targets to show. Set to NULL to show all the targets (default). Otherwise, you can supply symbols, a character vector, or tidyselect helpers like starts_with().
fields	Names of the fields, or columns, to show. Set to NULL to show all the fields (default). Otherwise, you can supply symbols, a character vector, or tidyselect helpers like starts_with(). Set to NULL to print all the fields. The name of the target is always included as the first column regardless of the selection. Possible fields are below. All of them can be set in tar_target(), tar_target_raw(), or tar_option_set().

- name: Name of the target.
- command: the R command that runs when the target builds.

- pattern: branching pattern of the target, if applicable.
- format: Storage format.
- iteration: Iteration mode for branching.
- error: Error mode, what to do when the target fails.
- memory: Memory mode, when to keep targets in memory.
- storage: Storage mode for high-performance computing scenarios.
- retrieval: Retrieval mode for high-performance computing scenarios.
- deployment: Where/whether to deploy the target in high-performance computing scenarios.
- resources: A list of target-specific resource requirements for tar_make_future().
- cue_mode: Cue mode from tar_cue().
- cue_depend: Depend cue from tar_cue().
- cue_expr: Command cue from tar_cue().
- cue_file: File cue from tar_cue().
- cue_format: Format cue from tar_cue().
- cue_iteration: Iteration cue from tar_cue().
- packages: List columns of packages loaded before building the target.
- library: List column of library paths to load the packages.

callr_function A function from callr to start a fresh clean R process to do the work. Set to NULL to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). callr_function needs to be NULL for interactive debugging, e.g. tar_option_set(debug = "your_target"). However, callr_function should not be NULL for serious reproducible work.

callr_arguments
A list of arguments to callr_function.

Value

A data frame of information about the targets in the pipeline. Rows appear in topological order (the order they will run without any influence from parallel computing or priorities).

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2),
    tar_target(m, z, pattern = map(z)),
    tar_target(c, z, pattern = cross(z))
  )
}, ask = FALSE)
tar_manifest()
```

```
tar_manifest(fields = c("name", "command"))
tar_manifest(fields = "command")
tar_manifest(fields = starts_with("cue"))
})
}
```

`tar_meta` *Read a project's metadata.*

Description

Read the metadata of all recorded targets and global objects.

Usage

```
tar_meta(
  names = NULL,
  fields = NULL,
  targets_only = FALSE,
  complete_only = FALSE
)
```

Arguments

`names`	Optional, names of the targets. If supplied, `tar_meta()` only returns metadata on these targets. You can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`. If `NULL`, all names are selected.
`fields`	Optional, names of columns/fields to select. If supplied, `tar_meta()` only returns the selected metadata columns. If `NULL`, all fields are selected. You can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`. The `name` column is always included first no matter what you select. Choices: • `name`: name of the target or global object. • `type`: type of the object: either `"function"` or `"object"` for imported global objects, and `"stem"`, `"branch"`, `"map"`, or `"cross"` for targets. • `data`: hash of the output data. • `command`: hash of the target's deparsed command. • `depend`: hash of the immediate upstream dependencies of the target. • `seed`: random number generator seed with which the target was built. A target's random number generator seed is a deterministic function of its name. In this way, each target runs with a reproducible seed so someone else running the same pipeline should get the same results, and no two targets in the same pipeline share the same seed. (Even dynamic branches have different names and thus different seeds.) You can recover the seed of a completed target with `tar_meta(your_target,seed)` and run `set.seed()` on the result to locally recreate the target's initial RNG state.

- `path`: A list column of paths to target data. Usually, each element is a single path, but there could be multiple paths per target for dynamic files (i.e. `tar_target(format = "file")`).
- `time`: hash of the maximum modification time stamp over all the files in `path`.
- `size`: hash of the sum of all the bytes of the files at `path`.
- `bytes`: total file size in bytes of all files in `path`.
- `format`: character, one of the admissible data storage formats. See the `format` argument in the `tar_target()` help file for details.
- `iteration`: character, either `"list"` or `"vector"` to describe the iteration and aggregation mode of the target. See the `iteration` argument in the `tar_target()` help file for details.
- `parent`: for branches, name of the parent pattern.
- `children`: list column, names of the children of targets that have them. These include buds of stems and branches of patterns.
- `seconds`: number of seconds it took to run the target.
- `warnings`: character string of warning messages from the last run of the target.
- `error`: character string of the error message if the target errored.

`targets_only` Logical, whether to just show information about targets or also return metadata on functions and other global objects.

`complete_only` Logical, whether to return only complete rows (no `NA` values).

Details

A metadata row only updates when the target is built. `tar_progress()` shows information on targets that are running. That is why the number of branches may disagree between `tar_meta()` and `tar_progress()` for actively running pipelines.

Value

A data frame with one row per target/object and the selected fields.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_meta()
tar_meta(starts_with("y_"))
})
}
```

tar_name *Get the name of the target currently running.*

Description

Get the name of the target currently running.

Usage

```
tar_name(default = "target")
```

Arguments

`default`	Character, value to return if `tar_name()` is called on its own outside a `targets` pipeline. Having a default lets users run things without `tar_make()`, which helps peel back layers of code and troubleshoot bugs.

Value

Character of length 1. If called inside a pipeline, `tar_name()` returns name of the target currently running. Otherwise, the return value is `default`.

Examples

```
tar_name()
tar_name(default = "custom_target_name")
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(tar_target(x, tar_name()), ask = FALSE)
tar_make()
tar_read(x)
})
}
```

tar_network *Return the vertices and edges of a pipeline dependency graph.*

Description

Analyze the pipeline defined in _targets.R and return the vertices and edges of the directed acyclic graph of dependency relationships.

Usage

```
tar_network(
  targets_only = FALSE,
  reporter = "silent",
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter)
)
```

Arguments

`targets_only`	Logical, whether to restrict the output to just targets (`FALSE`) or to also include imported global functions and objects.
`reporter`	Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices: • `"silent"`: print nothing. • `"forecast"`: print running totals of the checked and outdated targets found so far.
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.

Value

A list with two data frames: `vertices` and `edges`. The vertices data frame has one row per target with fields to denote the type of the target or object (stem, branch, map, cross, function, or object) and the target's status (up to date, outdated, running, canceled, or errored). The edges data frame has one row for every edge and columns `to` and `from` to mark the starting and terminating vertices.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_network(targets_only = TRUE)
})
}
```

tar_objects *List saved targets*

Description

List targets currently saved to _targets/objects/. Does not include dynamic files or cloud storage.

Usage

```
tar_objects(names = NULL)
```

Arguments

names	Optional tidyselect selector to return a tactical subset of target names. If NULL, all names are selected.

Value

Character vector of targets saved to _targets/objects/.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set(workspace = "x")
  list(tar_target(x, "value"))
}, ask = FALSE)
tar_make()
tar_objects()
tar_objects(starts_with("x"))
})
}
```

tar_option_get *Get a target option.*

Description

Get a target option. These options include default arguments to tar_target() such as packages, storage format, iteration type, and cue. Needs to be called before any calls to tar_target() in order to take effect.

Usage

```
tar_option_get(option)
```

Arguments

option	Character of length 1, name of an option to get. Must be one of the argument names of tar_option_set().

Details

This function goes well with tar_target_raw() when it comes to defining external interfaces on top of the targets package to create pipelines.

Value

Value of a target option.

Examples

```
tar_option_get("format") # default format before we set anything
tar_target(x, 1)$settings$format
tar_option_set(format = "fst_tbl") # new default format
tar_option_get("format")
tar_target(x, 1)$settings$format
tar_option_reset() # reset the format
tar_target(x, 1)$settings$format
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set(cue = tar_cue(mode = "always")) # All targets always run.
  list(tar_target(x, 1), tar_target(y, 2))
})
tar_make()
tar_make()
})
}
```

---

tar_option_reset *Reset all target options.*

Description

Reset all target options you previously chose with tar_option_set(). These options are mostly configurable default arguments to tar_target() and tar_target_raw().

Usage

```
tar_option_reset()
```

Value

Nothing.

Examples

```
tar_option_get("format") # default format before we set anything
tar_target(x, 1)$settings$format
tar_option_set(format = "fst_tbl") # new default format
tar_option_get("format")
tar_target(x, 1)$settings$format
tar_option_reset() # reset all options
tar_target(x, 1)$settings$format
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set(cue = tar_cue(mode = "always"))
  tar_option_reset() # Undo option above.
  list(tar_target(x, 1), tar_target(y, 2))
})
tar_make()
tar_make()
})
}
```

`tar_option_set` *Set target options.*

Description

Set target options, including default arguments to `tar_target()` such as packages, storage format, iteration type, and cue. See default options with `tar_option_get()`. To use `tar_option_set()` effectively, put it in your workflow's _targets.R script before calls to `tar_target()` or `tar_target_raw()`.

Usage

```
tar_option_set(
  tidy_eval = NULL,
  packages = NULL,
  imports = NULL,
  library = NULL,
  envir = NULL,
  format = NULL,
  iteration = NULL,
  error = NULL,
  memory = NULL,
  garbage_collection = NULL,
  deployment = NULL,
  priority = NULL,
  backoff = NULL,
  resources = NULL,
  storage = NULL,
  retrieval = NULL,
```

```
  cue = NULL,
  debug = NULL,
  workspaces = NULL
)
```

Arguments

tidy_eval	Logical, whether to enable tidy evaluation when interpreting command and pattern. If TRUE, you can use the "bang-bang" operator !! to programmatically insert the values of global objects.
packages	Character vector of packages to load right before the target builds. Use tar_option_set() to set packages globally for all subsequent targets you define.
imports	Character vector of package names to track global dependencies. For example, if you write tar_option_set(imports = "yourAnalysisPackage") early in _targets.R, then tar_make() will automatically rerun or skip targets in response to changes to the R functions and objects defined in yourAnalysisPackage. Does not account for low-level compiled code such as C/C++ or Fortran. If you supply multiple packages, e.g. tar_option_set(imports = c("p1","p2")), then the objects in p1 override the objects in p2 if there are name conflicts. Similarly, objects in tar_option_get("envir") override everything in tar_option_get("imports").
library	Character vector of library paths to try when loading packages.
envir	Environment containing functions and global objects used in the R commands to run targets. Defaults to the global environment. If envir is the global environment, all the promise objects are diffused before sending the data to parallel workers in tar_make_future() and tar_make_clustermq(), but otherwise the environment is unmodified. This behavior improves performance by decreasing the size of data sent to workers. If envir is not the global environment, then it should at least inherit from the global environment or base environment so targets can access attached packages. In the case of a non-global envir, targets attempts to remove potentially high memory objects that come directly from targets. That includes tar_target() objects of class "tar_target", as well as objects of class "tar_pipeline" or "tar_algorithm". This behavior improves performance by decreasing the size of data sent to workers. Package environments should not be assigned to envir. To include package objects as upstream dependencies in the pipeline, assign the package to the packages and imports arguments of tar_option_set().
format	Optional storage format for the target's return value. With the exception of format = "file", each target gets a file in _targets/objects, and each format is a different way to save and load this file. Possible formats: • "rds": Default, uses saveRDS() and readRDS(). Should work for most objects, but slow. • "qs": Uses qs::qsave() and qs::qread(). Should work for most objects, much faster than "rds". Optionally set the preset for qsave() through the resources argument, e.g. tar_target(...,resources = list(preset = "archive")). Requires the qs package (not installed by default).

- "feather": Uses arrow::write_feather() and arrow::read_feather() (version 2.0). Much faster than "rds", but the value must be a data frame. Optionally set compression and compression_level in arrow::write_feather() through the resources argument, e.g. tar_target(...,resources = list(compression = ...)). Requires the arrow package (not installed by default).
- "parquet": Uses arrow::write_parquet() and arrow::read_parquet() (version 2.0). Much faster than "rds", but the value must be a data frame. Optionally set compression and compression_level in arrow::write_parquet() through the resources argument, e.g. tar_target(...,resources = list(compression = ...)). Requires the arrow package (not installed by default).
- "fst": Uses fst::write_fst() and fst::read_fst(). Much faster than "rds", but the value must be a data frame. Optionally set the compression level for fst::write_fst() through the resources argument, e.g. tar_target(...,resources = list(compress = 100)). Requires the fst package (not installed by default).
- "fst_dt": Same as "fst", but the value is a data.table. Optionally set the compression level the same way as for "fst".
- "fst_tbl": Same as "fst", but the value is a tibble. Optionally set the compression level the same way as for "fst".
- "keras": Uses keras::save_model_hdf5() and keras::load_model_hdf5(). The value must be a Keras model. Requires the keras package (not installed by default).
- "torch": Uses torch::torch_save() and torch::torch_load(). The value must be an object from the torch package such as a tensor or neural network module. Requires the torch package (not installed by default).
- "file": A dynamic file. To use this format, the target needs to manually identify or save some data and return a character vector of paths to the data. (These paths must be existing files and nonempty directories.) Then, targets automatically checks those files and cues the appropriate build decisions if those files are out of date. Those paths must point to files or directories, and they must not contain characters | or *. All the files and directories you return must actually exist, or else targets will throw an error. (And if storage is "worker", targets will first stall out trying to wait for the file to arrive over a network file system.)
- "url": A dynamic input URL. It works like format = "file" except the return value of the target is a URL that already exists and serves as input data for downstream targets. Optionally supply a custom curl handle through the resources argument, e.g. tar_target(...,resources = list(handle = curl::new_handle())). The data file at the URL needs to have an ETag or a Last-Modified time stamp, or else the target will throw an error because it cannot track the data. Also, use extreme caution when trying to use format = "url" to track uploads. You must be absolutely certain the ETag and Last-Modified time stamp are fully updated and available by the time the target's command finishes running. targets makes no attempt to wait for the web server.
- "aws_rds", "aws_qs", "aws_parquet", "aws_fst", "aws_fst_dt", "aws_fst_tbl", "aws_keras": AWS-powered versions of the respective formats "rds",

"qs", etc. The only difference is that the data file is uploaded to the AWS S3 bucket you supply to resources. See the cloud computing chapter of the manual for details.

- "aws_file": arbitrary dynamic files on AWS S3. The target should return a path to a temporary local file, then targets will automatically upload this file to an S3 bucket and track it for you. Unlike format = "file", format = "aws_file" can only handle one single file, and that file must not be a directory. tar_read() and downstream targets download the file to _targets/scratch/ locally and return the path. _targets/scratch/ gets deleted at the end of tar_make(). Requires the same resources and other configuration details as the other AWS-powered formats. See the cloud computing chapter of the manual for details.

iteration — Character of length 1, name of the iteration mode of the target. Choices:

- "vector": branching happens with vctrs::vec_slice() and aggregation happens with vctrs::vec_c().
- "list", branching happens with [[]] and aggregation happens with list().
- "group": dplyr::group_by()-like functionality to branch over subsets of a data frame. The target's return value must be a data frame with a special tar_group column of consecutive integers from 1 through the number of groups. Each integer designates a group, and a branch is created for each collection of rows in a group. See the tar_group() function to see how you can create the special tar_group column with dplyr::group_by().

error — Character of length 1, what to do if the target runs into an error. If "stop", the whole pipeline stops and throws an error. If "continue", the error is recorded, but the pipeline keeps going.

memory — Character of length 1, memory strategy. If "persistent", the target stays in memory until the end of the pipeline (unless storage is "worker", in which case targets unloads the value from memory right after storing it in order to avoid sending copious data over a network). If "transient", the target gets unloaded after every new target completes. Either way, the target gets automatically loaded into memory whenever another target needs the value. For cloud-based dynamic files such as format = "aws_file", this memory policy applies to temporary local copies of the file in _targets/scratch/": "persistent" means they remain until the end of the pipeline, and "transient" means they get deleted from the file system as soon as possible. The former conserves bandwidth, and the latter conserves local storage.

garbage_collection — Logical, whether to run base::gc() just before the target runs.

deployment — Character of length 1, only relevant to tar_make_clustermq() and tar_make_future(). If "worker", the target builds on a parallel worker. If "main", the target builds on the host machine / process managing the pipeline.

priority — Numeric of length 1 between 0 and 1. Controls which targets get deployed first when multiple competing targets are ready simultaneously. Targets with priorities closer to 1 get built earlier (and polled earlier in tar_make_future()). Only applies to tar_make_future() and tar_make_clustermq() (not tar_make()). tar_make_future() with no extra settings is a drop-in replacement for tar_make() in this case.

backoff — Numeric of length 1, must be greater than or equal to 0.01. Maximum upper bound of the random polling interval for the priority queue (seconds). In high-performance computing (e.g. tar_make_clustermq() and tar_make_future()) it can be expensive to repeatedly poll the priority queue if no targets are ready to process. The number of seconds between polls is runif(1,0.01,max(backoff,0.01 * 1.5 ^ index)), where index is the number of consecutive polls so far that found no targets ready to skip or run. (If no target is ready, index goes up by 1. If a target is ready, index resets to 0. For more information on exponential, backoff, visit https://en.wikipedia.org/wiki/Exponential_backoff). Raising backoff is kinder to the CPU etc. but may incur delays in some instances.

resources — A named list of computing resources. Uses:

- Template file wildcards for future::future() in tar_make_future().
- Template file wildcards clustermq::workers() in tar_make_clustermq().
- Custom target-level future::plan(), e.g. resources = list(plan = future.callr::callr).
- Custom curl handle if format = "url", e.g. resources = list(handle = curl::new_handle(nobody = TRUE)). In custom handles, most users should manually set nobody = TRUE so targets does not download the entire file when it only needs to check the time stamp and ETag.
- Custom preset for qs::qsave() if format = "qs", e.g. resources = list(handle = "archive").
- Arguments compression and compression_level to arrow::write_feather() and arrow:write_parquet() if format is "feather", "parquet", "aws_feather", or "aws_parquet".
- Custom compression level for fst::write_fst() if format is "fst", "fst_dt", or "fst_tbl", e.g. resources = list(compress = 100).
- AWS bucket and prefix for the "aws_" formats, e.g. resources = list(bucket = "your-bucket",prefix = "folder/name"). bucket is required for AWS formats. See the cloud computing chapter of the manual for details.

storage — Character of length 1, only relevant to tar_make_clustermq() and tar_make_future(). If "main", the target's return value is sent back to the host machine and saved locally. If "worker", the worker saves the value.

retrieval — Character of length 1, only relevant to tar_make_clustermq() and tar_make_future(). If "main", the target's dependencies are loaded on the host machine and sent to the worker before the target builds. If "worker", the worker loads the targets dependencies.

cue — An optional object from tar_cue() to customize the rules that decide whether the target is up to date.

debug — Character vector of names of targets to run in debug mode. To use effectively, you must set callr_function = NULL and restart your R session just before running. You should also tar_make(), tar_make_clustermq(), or tar_make_future(). For any target mentioned in debug, targets will force the target to build locally (with tar_cue(mode = "always") and deployment = "main" in the settings) and pause in an interactive debugger to help you diagnose problems. This is like inserting a browser() statement at the beginning of the target's expression, but without invalidating any targets.

workspaces — Character vector of names of targets to save workspace files. Workspace files let you re-create a target's runtime environment in an interactive R session using `tar_workspace()`. `tar_workspace()` loads a target's random number generator seed and dependency objects as long as those target objects are still in the data store (usually _targets/objects/).

Value

Nothing.

Examples

```
tar_option_get("format") # default format before we set anything
tar_target(x, 1)$settings$format
tar_option_set(format = "fst_tbl") # new default format
tar_option_get("format")
tar_target(x, 1)$settings$format
tar_option_reset() # reset the format
tar_target(x, 1)$settings$format
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set(cue = tar_cue(mode = "always")) # All targets always run.
  list(tar_target(x, 1), tar_target(y, 2))
})
tar_make()
tar_make()
})
}
```

---

tar_outdated *Check which targets are outdated.*

Description

Checks for outdated targets in the pipeline, targets that will be rerun automatically if you call `tar_make()` or similar. See `tar_cue()` for the rules that decide whether a target needs to rerun.

Usage

```
tar_outdated(
  names = NULL,
  branches = FALSE,
  targets_only = TRUE,
  reporter = "silent",
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter)
)
```

Arguments

names	Names of the targets. tar_outdated() will check these targets and all upstream ancestors in the dependency graph. Set names to NULL to check/build all the targets (default). Otherwise, you can supply symbols, a character vector, or tidyselect helpers like starts_with().
branches	Logical, whether to include branch names. Including branches could get cumbersome for large pipelines. Individual branch names are still omitted when branch-specific information is not reliable: for example, when a pattern branches over an outdated target.
targets_only	Logical, whether to just restrict to targets or to include functions and other global objects from the environment created by running _targets.R.
reporter	Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices: • "silent": print nothing. • "forecast": print running totals of the checked and outdated targets found so far.
callr_function	A function from callr to start a fresh clean R process to do the work. Set to NULL to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). callr_function needs to be NULL for interactive debugging, e.g. tar_option_set(debug = "your_target"). However, callr_function should not be NULL for serious reproducible work.
callr_arguments	A list of arguments to callr_function.

Details

Requires that you define a pipeline with a _targets.R script in your working directory. (See tar_script() for details.)

Value

Names of the outdated targets.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(list(tar_target(x, 1 + 1)))
tar_outdated()
tar_script({
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_outdated()
```

```
})
}
```

`tar_path` *Identify the file path where a target will be stored.*

Description

Identify the file path where a target will be stored after the target finishes running in the pipeline.

Usage

```
tar_path(name = NULL, default = NA_character_)
```

Arguments

`name`	Symbol, name of a target. If `NULL`, `tar_path()` returns the path of the target currently running in a pipeline.
`default`	Character, value to return if `tar_path()` is called on its own outside a `targets` pipeline. Having a default lets users run things without `tar_make()`, which helps peel back layers of code and troubleshoot bugs.

Details

`tar_path(name = your_target)` just returns _targets/objects/your_target, the file path where `your_target` will be saved unless `format` is equal to `"file"` or any of the supported cloud-based storage formats. If you call `tar_path()` with no arguments while target `x` is running, the `name` argument defaults to the name of the target, so `tar_path()` returns _targets/objects/x.

Value

Character, file path to a hypothetical target.

Examples

```
tar_path()
tar_path(your_target)
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(tar_target(returns_path, tar_path()), ask = FALSE)
tar_make()
tar_read(returns_path)
})
}
```

`tar_pattern` *Emulate dynamic branching.*

Description

Emulate the dynamic branching process outside a pipeline. `tar_pattern()` can help you understand the overall branching structure that comes from the `pattern` argument of `tar_target()`.

Usage

```
tar_pattern(pattern, ..., seed = 0L)
```

Arguments

`pattern`	Function call with the pattern specification.
`...`	Named integers, each of length 1. Each name is the name of a dependency target, and each integer is the length of the target (number of branches or slices). Names must be unique.
`seed`	Integer of length 1, random number generator seed to emulate the pattern reproducibly. (The `sample()` pattern is random). In a real pipeline, the seed is automatically generated from the target name in deterministic fashion.

Details

Dynamic branching is a way to programmatically create multiple new targets based on the values of other targets, all while the pipeline is running. Use the `pattern` argument of `tar_target()` to get started. `pattern` accepts a function call composed of target names and any of the following patterns:

- `map()`: iterate over one or more targets in sequence.
- `cross()`: iterate over combinations of slices of targets.
- `head()`: restrict branching to the first few elements.
- `tail()`: restrict branching to the last few elements.
- `sample()`: restrict branching to a random subset of elements.

Value

A `tibble` showing the kinds of dynamic branches that `tar_target()` would create in a real pipeline with the given `pattern`. Each row is a dynamic branch, each column is a dependency target, and each element is the name of an upstream bud or branch that the downstream branch depends on. Buds are pieces of non-branching targets ("stems") and branches are pieces of patterns. The returned bud and branch names are not the actual ones you will see when you run the pipeline, but they do communicate the branching structure of the pattern.

Examples

```
# To use dynamic map for real in a pipeline,
# call map() in a target's pattern.
# The following code goes at the bottom of _targets.R.
list(
  tar_target(x, seq_len(2)),
  tar_target(y, head(letters, 2)),
  tar_target(dynamic, c(x, y), pattern = map(x, y)) # 2 branches
)
# Likewise for more complicated patterns.
list(
  tar_target(x, seq_len(2)),
  tar_target(y, head(letters, 2)),
  tar_target(z, head(LETTERS, 2)),
  tar_target(dynamic, c(x, y, z), pattern = cross(z, map(x, y))) #4 branches
)
# But you can emulate dynamic branching without running a pipeline
# in order to understand the patterns you are creating. Simply supply
# the pattern and the length of each dependency target.
# The returned data frame represents the branching structure of the pattern:
# One row per new branch, one column per dependency target, and
# one element per bud/branch in each dependency target.
tar_pattern(
  cross(x, map(y, z)),
  x = 2,
  y = 3,
  z = 3
)
tar_pattern(
  head(cross(x, map(y, z)), n = 2),
  x = 2,
  y = 3,
  z = 3
)
```

---

tar_pid *Get main process ID.*

Description

Get the process ID (PID) of the most recent main R process to orchestrate the targets of the current project.

Usage

```
tar_pid()
```

Details

The main process is the R process invoked by `tar_make()` or similar. If `callr_function` is not `NULL`, this is an external process, and the `pid` in the return value will not agree with `Sys.getpid()` in your current interactive session. The process may or may not be alive. You may want to check it with `ps::ps_is_running(ps::ps_handle(targets::tar_pid()))` before running another call to `tar_make()` for the same project.

Value

Integer with the process ID (PID) of the most recent main R process to orchestrate the targets of the current project.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
Sys.getpid()
tar_pid() # Different from the current PID.
})
}
```

---

tar_process *Get main process info.*

Description

Get info on the most recent main R process to orchestrate the targets of the current project.

Usage

```
tar_process(names = NULL)
```

Arguments

`names`	Optional, names of the data points to return. If supplied, `tar_process()` returns only the rows of the names you select. You can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`. If `NULL`, all names are selected.

Details

The main process is the R process invoked by `tar_make()` or similar. If `callr_function` is not `NULL`, this is an external process, and the `pid` in the return value will not agree with `Sys.getpid()` in your current interactive session. The process may or may not be alive. You may want to check the status with `tar_pid() %in% ps::ps_pids()` before running another call to `tar_make()` for the same project.

Value

A data frame with metadata on the most recent main R process to orchestrate the targets of the current project. The output includes the `pid` of the main process.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_process()
tar_process(pid)
})
}
```

---

tar_progress *Read the target progress of the latest run of the pipeline.*

Description

Read a project's target progress data for the most recent run of `tar_make()` or similar. Only the most recent record is shown.

Usage

```
tar_progress(names = NULL, fields = "progress")
```

Arguments

`names`	Optional, names of the targets. If supplied, `tar_progress()` only returns progress information on these targets. You can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
`fields`	Optional, names of progress data columns to read. Set to `NULL` to read all fields.

Value

A data frame with one row per target and the following columns:

- `name`: name of the target.
- `type`: type of target: `"stem"` for non-branching targets, `"pattern"` for dynamically branching targets, and `"branch"` for dynamic branches.
- `parent`: name of the target's parent. For branches, this is the name of the associated pattern. For other targets, the pattern is just itself.
- `branches`: number of dynamic branches of a pattern. 0 for non-patterns.
- `progress`: the most recent progress update of that target. Could be `"started"`, `"built"`, `"canceled"`, or `"errored"`.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_progress()
tar_progress(starts_with("y_"))
})
}
```

`tar_progress_branches` *Read the target progress of the latest run of the pipeline.*

Description

Read a project's target progress data for the most recent run of `tar_make()` or similar. Only the most recent record is shown.

Usage

```
tar_progress_branches(names = NULL, fields = NULL)
```

Arguments

`names`	Optional, names of the targets. If supplied, `tar_progress()` only returns progress information on these targets. You can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
`fields`	Optional, names of progress data columns to read. Set to `NULL` to read all fields.

Value

A data frame with one row per target per progress status and the following columns.

- `name`: name of the pattern.
- `progress`: progress status: `"started"`, `"built"`, `"cancelled"`, or `"errored"`.
- `branches`: number of branches in the progress category.
- `total`: total number of branches planned for the whole pattern. Values within the same pattern should all be equal.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(x, seq_len(2)),
    tar_target(y, x, pattern = map(x)),
    tar_target(z, stopifnot(y < 1.5), pattern = map(y))
  )
}, ask = FALSE)
try(tar_make())
tar_progress_branches()
})
}
```

`tar_prune` *Remove targets that are no longer part of the pipeline.*

Description

Remove target values from _targets/objects/ and target metadata from _targets/meta/meta for targets that are no longer part of the pipeline. Global objects and dynamic files outside the data store are unaffected. Also removes _targets/scratch/, which is only needed while `tar_make()`, `tar_make_clustermq()`, or `tar_make_future()` is running.

Usage

```
tar_prune(
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function)
)
```

Arguments

callr_function	A function from callr to start a fresh clean R process to do the work. Set to NULL to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). callr_function needs to be NULL for interactive debugging, e.g. tar_option_set(debug = "your_target"). However, callr_function should not be NULL for serious reproducible work.
callr_arguments	A list of arguments to callr_function.

Value

NULL except if callr_function = callr::r_bg(), in which case a handle to the callr background process is returned. Either way, the value is invisibly returned.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
}, ask = FALSE)
tar_make()
# Remove some targets from the pipeline.
tar_script(list(tar_target(y1, 1 + 1)), ask = FALSE)
# Keep only the remaining targets in the data store.
tar_prune()
})
}
```

---

tar_read *Read a target's value from storage.*

Description

Read a target's return value from its file in _targets/objects/. For dynamic files (i.e. format = "file") the paths are returned.

Usage

```
tar_read(name, branches = NULL, meta = tar_meta())
```

Arguments

name	Symbol, name of the target to read.
branches	Integer of indices of the branches to load if the target is a pattern.
meta	Data frame of metadata from tar_meta(). tar_read() with the default arguments can be inefficient for large pipelines because all the metadata is stored in a single file. However, if you call tar_meta() beforehand and supply it to the meta argument, then successive calls to tar_read() may run much faster.

Value

The target's return value from its file in _targets/objects/, or the paths to the custom files and directories if format = "file" was set.

Examples

```
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(list(tar_target(x, 1 + 1)), ask = FALSE)
tar_make()
tar_read(x)
})
```

---

tar_read_raw *Read a target's value from storage (raw version)*

Description

Like tar_read() except name is a character string. Do not use in knitr or R Markdown reports with tarchetypes::tar_knit() or tarchetypes::tar_render().

Usage

```
tar_read_raw(name, branches = NULL, meta = tar_meta())
```

Arguments

name	Character, name of the target to read.
branches	Integer of indices of the branches to load if the target is a pattern.
meta	Data frame of metadata from tar_meta(). tar_read() with the default arguments can be inefficient for large pipelines because all the metadata is stored in a single file. However, if you call tar_meta() beforehand and supply it to the meta argument, then successive calls to tar_read() may run much faster.

Value

The target's return value from its file in _targets/objects/, or the paths to the custom files and directories if format = "file" was set.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(list(tar_target(x, 1 + 1)), ask = FALSE)
tar_make()
tar_read_raw("x")
})
}
```

`tar_renv` *Set up package dependencies for compatibility with* `renv`

Description

Write package dependencies to a script file (by default, named _packages.R in the root project directory). Each package is written to a separate line as a standard `library()` call (e.g. `library(package)`) so `renv` can identify them automatically.

Usage

```
tar_renv(
 extras = c("bs4Dash", "clustermq", "future", "gt", "pingr", "rstudioapi", "shiny",
    "shinycssloaders", "shinyWidgets", "visNetwork"),
  path = "_packages.R",
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function)
)
```

Arguments

`extras`	Character vector of additional packages to declare as project dependencies.
`path`	Character of length 1, path to the script file to populate with `library()` calls.
`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.

Details

This function gets called for its side-effect, which writes package dependencies to a script for compatibility with `renv`. The generated file should **not** be edited by hand and will be overwritten each time `tar_renv()` is called.

The behavior of renv is to create and manage a project-local R library and keep a record of project dependencies in a file called renv.lock. To identify dependencies, renv crawls through code to find packages explicitly mentioned using library(), require(), or ::. However, targets manages packages in a way that hides dependencies from renv. tar_renv() finds package dependencies that would be otherwise hidden to renv because they are declared using the targets API. Thus, calling tar_renv this is only necessary if using tar_option_set() or tar_target() to use specialized storage formats or manage packages.

With the script written by tar_renv(), renv is able to crawl the file to identify package dependencies (with renv::dependencies()). tar_renv() only serves to make your targets project compatible with renv, it is still the users responsibility to call renv::init() and renv::snapshot() directly to initialize and manage a project-local R library. This allows your targets pipeline to have its own self-contained R library separate from your standard R library. See https://rstudio.github.io/renv/index.html for more information.

Value

Nothing, invisibly.

See Also

https://rstudio.github.io/renv/articles/renv.html

Examples

```
tar_dir({ # tar_dir() runs code from a temporary directory.
  tar_script({
    tar_option_set(packages = c("tibble", "qs"))
    list()
  }, ask = FALSE)
  tar_renv()
  writeLines(readLines("_packages.R"))
})
tar_option_reset()
```

---

tar_script *Write a* _targets.R *script to the current working directory.*

Description

The tar_script() function is a convenient way to create the required target script (_targets.R file) in the current working directory. It always overwrites the existing target script, and it requires you to be in the working directory where you intend to write the file, so be careful. See the "Target script" section for details.

Usage

```
tar_script(code = NULL, library_targets = TRUE, ask = NULL)
```

Arguments

code	R code to write to _targets.R. If NULL, an example target script is written instead.
library_targets	logical, whether to write a library(targets) line at the top of _targets.R automatically (recommended). If TRUE, you do not need to explicitly put library(targets) in code.
ask	Logical, whether to ask before writing if _targets.R already exists. If NULL, defaults to Sys.getenv("TAR_ASK"). (Set to "true" or "false" with Sys.setenv()). If ask and the TAR_ASK environment variable are both indeterminate, defaults to interactive().

Value

NULL (invisibly).

Target script

Every targets project requires a target script in the project root. The target script must always be named _targets.R. Functions tar_make() and friends look for _targets.R in the current working directory and use it to set up the pipeline. Every _targets.R file should run the following steps in the order below: 1. Package: load the targets package. This step is automatically inserted at the top of _targets.R files produced by tar_script() if library_targets is TRUE, so you do not need to explicitly include it in code. 1. Globals: load custom functions and global objects into memory. Usually, this section is a bunch of calls to source() that run scripts defining user-defined functions. These functions support the R commands of the targets. 2. Options: call tar_option_set() to set defaults for targets-specific settings such as the names of required packages. Even if you have no specific options to set, it is still recommended to call tar_option_set() in order to register the proper environment. 3. Targets: define one or more target objects using tar_target(). 4. Pipeline: call list() to bring the targets from (3) together in a pipeline object. Every _targets.R script must return a pipeline object, which usually means ending with a call to list(). In practice, (3) and (4) can be combined together in the same function call.

Examples

```
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script() # Writes an example target script.
# Writes a user-defined target script:
tar_script({
  x <- tar_target(x, 1 + 1)
  tar_option_set()
  list(x)
}, ask = FALSE)
writeLines(readLines("_targets.R"))
})
```

tar_seed *Get the random number generator seed of the target currently running.*

Description

Get the random number generator seed of the target currently running.

Usage

```
tar_seed(default = 1L)
```

Arguments

default	Integer, value to return if `tar_seed()` is called on its own outside a `targets` pipeline. Having a default lets users run things without `tar_make()`, which helps peel back layers of code and troubleshoot bugs.

Details

A target's random number generator seed is a deterministic function of its name. In this way, each target runs with a reproducible seed so someone else running the same pipeline should get the same results, and no two targets in the same pipeline share the same seed. (Even dynamic branches have different names and thus different seeds.) You can retrieve the seed of a completed target with `tar_meta(your_target,seed)` and run `set.seed()` on the result to locally recreate the target's initial RNG state.

Value

Integer of length 1. If invoked inside a `targets` pipeline, the return value is the seed of the target currently running, which is a deterministic function of the target name. Otherwise, the return value is `default`.

Examples

```
tar_seed()
tar_seed(default = 123L)
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(tar_target(returns_seed, tar_seed()), ask = FALSE)
tar_make()
tar_read(returns_seed)
})
}
```

`tar_sitrep` *Show the cue-by-cue status of each target.*

Description

For each target, report which cues are activated. Except for the `never` cue, the target will rerun in `tar_make()` if any cue is activated. The target is suppressed if the `never` cue is `TRUE`. See `tar_cue()` for details.

Usage

```
tar_sitrep(
  names = NULL,
  fields = NULL,
  reporter = "silent",
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter)
)
```

Arguments

`names`	Optional, names of the targets. If supplied, `tar_sitrep()` only returns metadata on these targets. You can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
`fields`	Optional, names of columns/fields to select. If supplied, `tar_sitrep()` only returns the selected metadata columns. You can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`. The `name` column is always included first no matter what you select. Choices: • `name`: name of the target or global object. • `record`: Whether the `record` cue is activated: `TRUE` if the target is not in the metadata (`tar_meta()`), or if the target errored during the last `tar_make()`, or if the class of the target changed. • `always`: Whether `mode` in `tar_cue()` is `"always"`. If `TRUE`, `tar_make()` always runs the target. • `never`: Whether `mode` in `tar_cue()` is `"never"`. If `TRUE`, `tar_make()` will only run if the `record` cue activates. • `command`: Whether the target's command changed since last time. Always `TRUE` if the `record` cue is activated. Otherwise, always `FALSE` if the `command` cue is suppressed. • `depend`: Whether the data/output of at least one of the target's dependencies changed since last time. Dependencies are targets, functions, and global objects directly upstream. Call `tar_outdated(targets_only = FALSE)` or `tar_visnetwork(targets_only = FALSE)` to see exactly which dependencies are outdated. Always `NA` if the `record` cue is activated. Otherwise, always `FALSE` if the `depend` cue is suppressed.

- `format`: Whether the storage format of the target is different from last time. Always `NA` if the `record` cue is activated. Otherwise, always `FALSE` if the `format` cue is suppressed.
- `iteration`: Whether the iteration mode of the target is different from last time. Always `NA` if the `record` cue is activated. Otherwise, always `FALSE` if the `iteration` cue is suppressed.
- `file`: Whether the file(s) with the target's return value are missing or different from last time. Always `NA` if the `record` cue is activated. Otherwise, always `FALSE` if the `file` cue is suppressed.

`reporter` Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices:

- `"silent"`: print nothing.
- `"forecast"`: print running totals of the checked and outdated targets found so far.

`callr_function` A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.

`callr_arguments`
A list of arguments to `callr_function`.

Details

Caveats:

- `tar_cue()` allows you to change/suppress cues, so the return value will depend on the settings you supply to `tar_cue()`.
- If a pattern tries to branches over a target that does not exist in storage, then the branches are omitted from the output.
- `tar_sitrep()` is myopic. It only considers what happens to the immediate target and its immediate upstream dependencies, and it makes no attempt to propagate invalidation downstream.

Value

A data frame with one row per target/object and one column per cue. Each element is a logical to indicate whether the cue is activated for the target. See the `field` argument in this help file for details.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  list(
    tar_target(x, seq_len(2)),
```

```
    tar_target(y, 2 * x, pattern = map(x))
  )
}, ask = FALSE)
tar_make()
tar_sitrep()
tar_meta(starts_with("y_"))
})
}
```

`tar_target` *Declare a target.*

Description

A target is a single step of computation in a pipeline. It runs an R command and returns a value. This value gets treated as an R object that can be used by the commands of targets downstream. Targets that are already up to date are skipped. See the user manual for more details.

Usage

```
tar_target(
  name,
  command,
  pattern = NULL,
  tidy_eval = targets::tar_option_get("tidy_eval"),
  packages = targets::tar_option_get("packages"),
  library = targets::tar_option_get("library"),
  format = targets::tar_option_get("format"),
  iteration = targets::tar_option_get("iteration"),
  error = targets::tar_option_get("error"),
  memory = targets::tar_option_get("memory"),
  garbage_collection = targets::tar_option_get("garbage_collection"),
  deployment = targets::tar_option_get("deployment"),
  priority = targets::tar_option_get("priority"),
  resources = targets::tar_option_get("resources"),
  storage = targets::tar_option_get("storage"),
  retrieval = targets::tar_option_get("retrieval"),
  cue = targets::tar_option_get("cue")
)
```

Arguments

`name`	Symbol, name of the target. Subsequent targets can refer to this name symbolically to induce a dependency relationship: e.g. `tar_target(downstream_target,f(upstream_target)` is a target named `downstream_target` which depends on a target `upstream_target` and a function `f()`. In addition, a target's name determines its random number generator seed. In this way, each target runs with a reproducible seed so someone else running the same pipeline should get the same results, and no two

targets in the same pipeline share the same seed. (Even dynamic branches have different names and thus different seeds.) You can recover the seed of a completed target with `tar_meta(your_target,seed)` and run `set.seed()` on the result to locally recreate the target's initial RNG state.

`command` R code to run the target.

`pattern` Language to define branching for a target. For example, in a pipeline with numeric vector targets `x` and `y`, `tar_target(z,x + y,pattern = map(x,y))` implicitly defines branches of `z` that each compute `x[1] + y[1]`, `x[2] + y[2]`, and so on. See the user manual for details.

`tidy_eval` Logical, whether to enable tidy evaluation when interpreting `command` and `pattern`. If `TRUE`, you can use the "bang-bang" operator !! to programmatically insert the values of global objects.

`packages` Character vector of packages to load right before the target builds. Use `tar_option_set()` to set packages globally for all subsequent targets you define.

`library` Character vector of library paths to try when loading `packages`.

`format` Optional storage format for the target's return value. With the exception of `format = "file"`, each target gets a file in _targets/objects, and each format is a different way to save and load this file. Possible formats:

- `"rds"`: Default, uses `saveRDS()` and `readRDS()`. Should work for most objects, but slow.
- `"qs"`: Uses `qs::qsave()` and `qs::qread()`. Should work for most objects, much faster than `"rds"`. Optionally set the preset for `qsave()` through the `resources` argument, e.g. `tar_target(...,resources = list(preset = "archive"))`. Requires the `qs` package (not installed by default).
- `"feather"`: Uses `arrow::write_feather()` and `arrow::read_feather()` (version 2.0). Much faster than `"rds"`, but the value must be a data frame. Optionally set `compression` and `compression_level` in `arrow::write_feather()` through the `resources` argument, e.g. `tar_target(...,resources = list(compression = ...))`. Requires the `arrow` package (not installed by default).
- `"parquet"`: Uses `arrow::write_parquet()` and `arrow::read_parquet()` (version 2.0). Much faster than `"rds"`, but the value must be a data frame. Optionally set `compression` and `compression_level` in `arrow::write_parquet()` through the `resources` argument, e.g. `tar_target(...,resources = list(compression = ...))`. Requires the `arrow` package (not installed by default).
- `"fst"`: Uses `fst::write_fst()` and `fst::read_fst()`. Much faster than `"rds"`, but the value must be a data frame. Optionally set the compression level for `fst::write_fst()` through the `resources` argument, e.g. `tar_target(...,resources = list(compress = 100))`. Requires the `fst` package (not installed by default).
- `"fst_dt"`: Same as `"fst"`, but the value is a `data.table`. Optionally set the compression level the same way as for `"fst"`.
- `"fst_tbl"`: Same as `"fst"`, but the value is a `tibble`. Optionally set the compression level the same way as for `"fst"`.
- `"keras"`: Uses `keras::save_model_hdf5()` and `keras::load_model_hdf5()`. The value must be a Keras model. Requires the `keras` package (not installed by default).

- "torch": Uses torch::torch_save() and torch::torch_load(). The value must be an object from the torch package such as a tensor or neural network module. Requires the torch package (not installed by default).
- "file": A dynamic file. To use this format, the target needs to manually identify or save some data and return a character vector of paths to the data. (These paths must be existing files and nonempty directories.) Then, targets automatically checks those files and cues the appropriate build decisions if those files are out of date. Those paths must point to files or directories, and they must not contain characters | or *. All the files and directories you return must actually exist, or else targets will throw an error. (And if storage is "worker", targets will first stall out trying to wait for the file to arrive over a network file system.)
- "url": A dynamic input URL. It works like format = "file" except the return value of the target is a URL that already exists and serves as input data for downstream targets. Optionally supply a custom curl handle through the resources argument, e.g. tar_target(...,resources = list(handle = curl::new_handle())). The data file at the URL needs to have an ETag or a Last-Modified time stamp, or else the target will throw an error because it cannot track the data. Also, use extreme caution when trying to use format = "url" to track uploads. You must be absolutely certain the ETag and Last-Modified time stamp are fully updated and available by the time the target's command finishes running. targets makes no attempt to wait for the web server.
- "aws_rds", "aws_qs", "aws_parquet", "aws_fst", "aws_fst_dt", "aws_fst_tbl", "aws_keras": AWS-powered versions of the respective formats "rds", "qs", etc. The only difference is that the data file is uploaded to the AWS S3 bucket you supply to resources. See the cloud computing chapter of the manual for details.
- "aws_file": arbitrary dynamic files on AWS S3. The target should return a path to a temporary local file, then targets will automatically upload this file to an S3 bucket and track it for you. Unlike format = "file", format = "aws_file" can only handle one single file, and that file must not be a directory. tar_read() and downstream targets download the file to _targets/scratch/ locally and return the path. _targets/scratch/ gets deleted at the end of tar_make(). Requires the same resources and other configuration details as the other AWS-powered formats. See the cloud computing chapter of the manual for details.

iteration Character of length 1, name of the iteration mode of the target. Choices:

- "vector": branching happens with vctrs::vec_slice() and aggregation happens with vctrs::vec_c().
- "list", branching happens with [[]] and aggregation happens with list().
- "group": dplyr::group_by()-like functionality to branch over subsets of a data frame. The target's return value must be a data frame with a special tar_group column of consecutive integers from 1 through the number of groups. Each integer designates a group, and a branch is created for each collection of rows in a group. See the tar_group() function to see how you can create the special tar_group column with dplyr::group_by().

`error` Character of length 1, what to do if the target runs into an error. If `"stop"`, the whole pipeline stops and throws an error. If `"continue"`, the error is recorded, but the pipeline keeps going.

`memory` Character of length 1, memory strategy. If `"persistent"`, the target stays in memory until the end of the pipeline (unless `storage` is `"worker"`, in which case `targets` unloads the value from memory right after storing it in order to avoid sending copious data over a network). If `"transient"`, the target gets unloaded after every new target completes. Either way, the target gets automatically loaded into memory whenever another target needs the value. For cloud-based dynamic files such as `format = "aws_file"`, this memory policy applies to temporary local copies of the file in _targets/scratch/": `"persistent"` means they remain until the end of the pipeline, and `"transient"` means they get deleted from the file system as soon as possible. The former conserves bandwidth, and the latter conserves local storage.

`garbage_collection` Logical, whether to run `base::gc()` just before the target runs.

`deployment` Character of length 1, only relevant to `tar_make_clustermq()` and `tar_make_future()`. If `"worker"`, the target builds on a parallel worker. If `"main"`, the target builds on the host machine / process managing the pipeline.

`priority` Numeric of length 1 between 0 and 1. Controls which targets get deployed first when multiple competing targets are ready simultaneously. Targets with priorities closer to 1 get built earlier (and polled earlier in `tar_make_future()`). Only applies to `tar_make_future()` and `tar_make_clustermq()` (not `tar_make()`). `tar_make_future()` with no extra settings is a drop-in replacement for `tar_make()` in this case.

`resources` A named list of computing resources. Uses:

- Template file wildcards for `future::future()` in `tar_make_future()`.
- Template file wildcards `clustermq::workers()` in `tar_make_clustermq()`.
- Custom target-level `future::plan()`, e.g. `resources = list(plan = future.callr::callr)`.
- Custom `curl` handle if `format = "url"`, e.g. `resources = list(handle = curl::new_handle(nobody = TRUE))`. In custom handles, most users should manually set `nobody = TRUE` so `targets` does not download the entire file when it only needs to check the time stamp and ETag.
- Custom preset for `qs::qsave()` if `format = "qs"`, e.g. `resources = list(handle = "archive")`.
- Arguments `compression` and `compression_level` to `arrow::write_feather()` and `arrow:write_parquet()` if `format` is `"feather"`, `"parquet"`, `"aws_feather"`, or `"aws_parquet"`.
- Custom compression level for `fst::write_fst()` if `format` is `"fst"`, `"fst_dt"`, or `"fst_tbl"`, e.g. `resources = list(compress = 100)`.
- AWS bucket and prefix for the `"aws_"` formats, e.g. `resources = list(bucket = "your-bucket",prefix = "folder/name")`. `bucket` is required for AWS formats. See the cloud computing chapter of the manual for details.

`storage` Character of length 1, only relevant to `tar_make_clustermq()` and `tar_make_future()`. If `"main"`, the target's return value is sent back to the host machine and saved locally. If `"worker"`, the worker saves the value.

retrieval	Character of length 1, only relevant to tar_make_clustermq() and tar_make_future(). If "main", the target's dependencies are loaded on the host machine and sent to the worker before the target builds. If "worker", the worker loads the targets dependencies.
cue	An optional object from tar_cue() to customize the rules that decide whether the target is up to date.

Value

A target object. Users should not modify these directly, just feed them to list() in your _targets.R file.

Examples

```
# Defining targets does not run them.
data <- tar_target(target_name, get_data(), packages = "tidyverse")
analysis <- tar_target(analysis, analyze(x), pattern = map(x))
# Pipelines accept targets.
pipeline <- list(data, analysis)
# Tidy evaluation
tar_option_set(envir = environment())
n_rows <- 30L
data <- tar_target(target_name, get_data(!!n_rows))
print(data)
# Disable tidy evaluation:
data <- tar_target(target_name, get_data(!!n_rows), tidy_eval = FALSE)
print(data)
tar_option_reset()
# In a pipeline:
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(tar_target(x, 1 + 1), ask = FALSE)
tar_make()
tar_read(x)
})
}
```

---

tar_target_raw *Define a target using unrefined names and language objects.*

Description

tar_target_raw() is just like tar_target() except it avoids non-standard evaluation for the arguments: name is a character string, command and pattern are language objects, and there is no tidy_eval argument. Use tar_target_raw() instead of tar_target() if you are creating entire batches of targets programmatically (metaprogramming, static branching).

Usage

```
tar_target_raw(
  name,
  command,
  pattern = NULL,
  packages = targets::tar_option_get("packages"),
  library = targets::tar_option_get("library"),
  deps = NULL,
  string = NULL,
  format = targets::tar_option_get("format"),
  iteration = targets::tar_option_get("iteration"),
  error = targets::tar_option_get("error"),
  memory = targets::tar_option_get("memory"),
  garbage_collection = targets::tar_option_get("garbage_collection"),
  deployment = targets::tar_option_get("deployment"),
  priority = targets::tar_option_get("priority"),
  resources = targets::tar_option_get("resources"),
  storage = targets::tar_option_get("storage"),
  retrieval = targets::tar_option_get("retrieval"),
  cue = targets::tar_option_get("cue")
)
```

Arguments

name	Character of length 1, name of the target. Subsequent targets can refer to this name symbolically to induce a dependency relationship: e.g. `tar_target(downstream_target,f(upstr` is a target named `downstream_target` which depends on a target `upstream_target` and a function `f()`. In addition, a target's name determines its random number generator seed. In this way, each target runs with a reproducible seed so someone else running the same pipeline should get the same results, and no two targets in the same pipeline share the same seed. (Even dynamic branches have different names and thus different seeds.) You can recover the seed of a completed target with `tar_meta(your_target,seed)` and run `set.seed()` on the result to locally recreate the target's initial RNG state.
command	Similar to the `command` argument of `tar_target()` except the object must already be an expression instead of informally quoted code. `base::expression()` and `base::quote()` can produce such objects.
pattern	Similar to the `pattern` argument of `tar_target()` except the object must already be an expression instead of informally quoted code. `base::expression()` and `base::quote()` can produce such objects.
packages	Character vector of packages to load right before the target builds. Use `tar_option_set()` to set packages globally for all subsequent targets you define.
library	Character vector of library paths to try when loading `packages`.
deps	Optional character vector of the adjacent upstream dependencies of the target, including targets and global objects. If `NULL`, dependencies are resolved automatically as usual.

`string` Optional string representation of the command. Internally, the string gets hashed to check if the command changed since last run, which helps `targets` decide whether the target is up to date. External interfaces can take control of `string` to ignore changes in certain parts of the command. If `NULL`, the strings is just deparsed from `command` (default).

`format` Optional storage format for the target's return value. With the exception of `format = "file"`, each target gets a file in _targets/objects, and each format is a different way to save and load this file. Possible formats:

- `"rds"`: Default, uses `saveRDS()` and `readRDS()`. Should work for most objects, but slow.
- `"qs"`: Uses `qs::qsave()` and `qs::qread()`. Should work for most objects, much faster than `"rds"`. Optionally set the preset for `qsave()` through the `resources` argument, e.g. `tar_target(...,resources = list(preset = "archive"))`. Requires the `qs` package (not installed by default).
- `"feather"`: Uses `arrow::write_feather()` and `arrow::read_feather()` (version 2.0). Much faster than `"rds"`, but the value must be a data frame. Optionally set `compression` and `compression_level` in `arrow::write_feather()` through the `resources` argument, e.g. `tar_target(...,resources = list(compression = ...))`. Requires the `arrow` package (not installed by default).
- `"parquet"`: Uses `arrow::write_parquet()` and `arrow::read_parquet()` (version 2.0). Much faster than `"rds"`, but the value must be a data frame. Optionally set `compression` and `compression_level` in `arrow::write_parquet()` through the `resources` argument, e.g. `tar_target(...,resources = list(compression = ...))`. Requires the `arrow` package (not installed by default).
- `"fst"`: Uses `fst::write_fst()` and `fst::read_fst()`. Much faster than `"rds"`, but the value must be a data frame. Optionally set the compression level for `fst::write_fst()` through the `resources` argument, e.g. `tar_target(...,resources = list(compress = 100))`. Requires the `fst` package (not installed by default).
- `"fst_dt"`: Same as `"fst"`, but the value is a `data.table`. Optionally set the compression level the same way as for `"fst"`.
- `"fst_tbl"`: Same as `"fst"`, but the value is a `tibble`. Optionally set the compression level the same way as for `"fst"`.
- `"keras"`: Uses `keras::save_model_hdf5()` and `keras::load_model_hdf5()`. The value must be a Keras model. Requires the `keras` package (not installed by default).
- `"torch"`: Uses `torch::torch_save()` and `torch::torch_load()`. The value must be an object from the `torch` package such as a tensor or neural network module. Requires the `torch` package (not installed by default).
- `"file"`: A dynamic file. To use this format, the target needs to manually identify or save some data and return a character vector of paths to the data. (These paths must be existing files and nonempty directories.) Then, `targets` automatically checks those files and cues the appropriate build decisions if those files are out of date. Those paths must point to files or directories, and they must not contain characters `|` or `*`. All the files and directories you return must actually exist, or else `targets` will throw an

error. (And if `storage` is `"worker"`, `targets` will first stall out trying to wait for the file to arrive over a network file system.)

- `"url"`: A dynamic input URL. It works like `format = "file"` except the return value of the target is a URL that already exists and serves as input data for downstream targets. Optionally supply a custom `curl` handle through the `resources` argument, e.g. `tar_target(...,resources = list(handle = curl::new_handle()))`. The data file at the URL needs to have an ETag or a Last-Modified time stamp, or else the target will throw an error because it cannot track the data. Also, use extreme caution when trying to use `format = "url"` to track uploads. You must be absolutely certain the ETag and Last-Modified time stamp are fully updated and available by the time the target's command finishes running. `targets` makes no attempt to wait for the web server.
- `"aws_rds"`, `"aws_qs"`, `"aws_parquet"`, `"aws_fst"`, `"aws_fst_dt"`, `"aws_fst_tbl"`, `"aws_keras"`: AWS-powered versions of the respective formats `"rds"`, `"qs"`, etc. The only difference is that the data file is uploaded to the AWS S3 bucket you supply to `resources`. See the cloud computing chapter of the manual for details.
- `"aws_file"`: arbitrary dynamic files on AWS S3. The target should return a path to a temporary local file, then `targets` will automatically upload this file to an S3 bucket and track it for you. Unlike `format = "file"`, `format = "aws_file"` can only handle one single file, and that file must not be a directory. `tar_read()` and downstream targets download the file to _targets/scratch/ locally and return the path. _targets/scratch/ gets deleted at the end of `tar_make()`. Requires the same `resources` and other configuration details as the other AWS-powered formats. See the cloud computing chapter of the manual for details.

`iteration` Character of length 1, name of the iteration mode of the target. Choices:

- `"vector"`: branching happens with `vctrs::vec_slice()` and aggregation happens with `vctrs::vec_c()`.
- `"list"`, branching happens with [[]] and aggregation happens with `list()`.
- `"group"`: `dplyr::group_by()`-like functionality to branch over subsets of a data frame. The target's return value must be a data frame with a special `tar_group` column of consecutive integers from 1 through the number of groups. Each integer designates a group, and a branch is created for each collection of rows in a group. See the `tar_group()` function to see how you can create the special `tar_group` column with `dplyr::group_by()`.

`error` Character of length 1, what to do if the target runs into an error. If `"stop"`, the whole pipeline stops and throws an error. If `"continue"`, the error is recorded, but the pipeline keeps going.

`memory` Character of length 1, memory strategy. If `"persistent"`, the target stays in memory until the end of the pipeline (unless `storage` is `"worker"`, in which case `targets` unloads the value from memory right after storing it in order to avoid sending copious data over a network). If `"transient"`, the target gets unloaded after every new target completes. Either way, the target gets automatically loaded into memory whenever another target needs the value. For cloud-based dynamic files such as `format = "aws_file"`, this memory policy

applies to temporary local copies of the file in _targets/scratch/": "persistent" means they remain until the end of the pipeline, and "transient" means they get deleted from the file system as soon as possible. The former conserves bandwidth, and the latter conserves local storage.

garbage_collection
: Logical, whether to run base::gc() just before the target runs.

deployment
: Character of length 1, only relevant to tar_make_clustermq() and tar_make_future(). If "worker", the target builds on a parallel worker. If "main", the target builds on the host machine / process managing the pipeline.

priority
: Numeric of length 1 between 0 and 1. Controls which targets get deployed first when multiple competing targets are ready simultaneously. Targets with priorities closer to 1 get built earlier (and polled earlier in tar_make_future()). Only applies to tar_make_future() and tar_make_clustermq() (not tar_make()). tar_make_future() with no extra settings is a drop-in replacement for tar_make() in this case.

resources
: A named list of computing resources. Uses:
  - Template file wildcards for future::future() in tar_make_future().
  - Template file wildcards clustermq::workers() in tar_make_clustermq().
  - Custom target-level future::plan(), e.g. resources = list(plan = future.callr::callr).
  - Custom curl handle if format = "url", e.g. resources = list(handle = curl::new_handle(nobody = TRUE)). In custom handles, most users should manually set nobody = TRUE so targets does not download the entire file when it only needs to check the time stamp and ETag.
  - Custom preset for qs::qsave() if format = "qs", e.g. resources = list(handle = "archive").
  - Arguments compression and compression_level to arrow::write_feather() and arrow:write_parquet() if format is "feather", "parquet", "aws_feather", or "aws_parquet".
  - Custom compression level for fst::write_fst() if format is "fst", "fst_dt", or "fst_tbl", e.g. resources = list(compress = 100).
  - AWS bucket and prefix for the "aws_" formats, e.g. resources = list(bucket = "your-bucket",prefix = "folder/name"). bucket is required for AWS formats. See the cloud computing chapter of the manual for details.

storage
: Character of length 1, only relevant to tar_make_clustermq() and tar_make_future(). If "main", the target's return value is sent back to the host machine and saved locally. If "worker", the worker saves the value.

retrieval
: Character of length 1, only relevant to tar_make_clustermq() and tar_make_future(). If "main", the target's dependencies are loaded on the host machine and sent to the worker before the target builds. If "worker", the worker loads the targets dependencies.

cue
: An optional object from tar_cue() to customize the rules that decide whether the target is up to date.

Value

A target object. Users should not modify these directly, just feed them to list() in your _targets.R file.

Examples

```
  # The following are equivalent.
  y <- tar_target(y, sqrt(x), pattern = map(x))
  y <- tar_target_raw("y", expression(sqrt(x)), expression(map(x)))
  # Programmatically create a chain of interdependent targets
  target_list <- lapply(seq_len(4), function(i) {
    tar_target_raw(
      letters[i + 1],
      substitute(do_something(x), env = list(x = as.symbol(letters[i])))
    )
  })
  print(target_list[[1]])
  print(target_list[[2]])
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(tar_target_raw("x", quote(1 + 1)), ask = FALSE)
tar_make()
tar_read(x)
})
}
```

`tar_test` *Test code in a temporary directory.*

Description

Runs a `test_that()` unit test inside a temporary directory to avoid writing to the user's file space. This helps ensure compliance with CRAN policies. Also isolates `tar_option_set()` options and environment variables specific to `targets` and skips the test on Solaris. Useful for writing tests for targetopia packages (extensions to `targets` tailored to specific use cases).

Usage

```
tar_test(label, code)
```

Arguments

`label`	Character of length 1, label for the test.
`code`	User-defined code for the test.

Value

`NULL` (invisibly).

Examples

```
tar_test("example test", {
  testing_variable_cafecfcb <- "only defined inside tar_test()"
  file.create("only_exists_in_tar_test")
})
exists("testing_variable_cafecfcb")
file.exists("only_exists_in_tar_test")
```

`tar_traceback` *Get a target's traceback*

Description

If a target ran with `error = "workspace"` and errored out, `tar_traceback()` returns its traceback. The workspace file must exist. For more information, see `tar_workspace()`.

Usage

```
tar_traceback(name, envir = parent.frame(), packages = TRUE, source = TRUE)
```

Arguments

`name`	Symbol, name of the target whose workspace to read.
`envir`	Environment in which to put the objects.
`packages`	Logical, whether to load the required packages of the target.
`source`	Logical, whether to run _targets.R to load user-defined global object dependencies into `envir`. If `TRUE`, then `envir` should either be the global environment or inherit from the global environment.

Value

Character vector, the traceback of a failed target if it exists

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tmp <- sample(1)
tar_script({
  tar_option_set(error = "workspace")
  list(
    tar_target(x, "loaded"),
    tar_target(y, stop(x))
  )
}, ask = FALSE)
try(tar_make())
tar_traceback(y)
})
}
```

`tar_validate` *Validate a pipeline of targets.*

Description

Inspect the pipeline for issues and throw an error or warning if a problem is detected.

Usage

```
tar_validate(
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function)
)
```

Arguments

`callr_function`	A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
`callr_arguments`	A list of arguments to `callr_function`.

Value

`NULL` except if `callr_function = callr::r_bg()`, in which case a handle to the `callr` background process is returned. Either way, the value is invisibly returned.

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script(list(tar_target(x, 1 + 1)), ask = FALSE)
tar_validate()
})
}
```

`tar_visnetwork` *Visualize an abridged fast dependency graph.*

Description

Analyze the pipeline defined in _targets.R and visualize the directed acyclic graph of targets and imported global functions and objects.

Usage

```
tar_visnetwork(
  targets_only = FALSE,
  allow = NULL,
  exclude = ".Random.seed",
  outdated = TRUE,
  label = NULL,
  level_separation = NULL,
  reporter = "silent",
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function)
)
```

Arguments

`targets_only`	Logical, whether to restrict the output to just targets (`FALSE`) or to also include imported global functions and objects.
`allow`	Optional, define the set of allowable vertices in the graph. Set to `NULL` to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
`exclude`	Optional, define the set of exclude vertices from the graph. Set to `NULL` to exclude no vertices. Otherwise, you can supply symbols, a character vector, or `tidyselect` helpers like `starts_with()`.
`outdated`	Logical, whether to show colors to distinguish outdated targets from up-to-date targets. (Global functions and objects still show these colors.) Looking for outdated targets takes a lot of time for large pipelines with lots of branches, and setting `outdated` to `FALSE` is a nice way to speed up the graph if you only want to see dependency relationships and build progress.
`label`	Character vector of one or more aesthetics to add to the vertex labels. Can contain `"time"` to show total runtime, `"size"` to show total storage size, or `"branches"` to show the number of branches in each pattern. You can choose multiple aesthetics at once, e.g. `label = c("time", "branches")`. All are disabled by default because they clutter the graph.
`level_separation`	Numeric of length 1, `levelSeparation` argument of `visNetwork::visHierarchicalLayout()`. Controls the distance between hierarchical levels. Consider changing the value

if the aspect ratio of the graph is far from 1. If `level_separation` is `NULL`, the `levelSeparation` argument of `visHierarchicalLayout()` defaults to `150`.

`reporter` Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices:

- `"silent"`: print nothing.
- `"forecast"`: print running totals of the checked and outdated targets found so far.

`callr_function` A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.

`callr_arguments`
A list of arguments to `callr_function`.

Value

A `visNetwork` HTML widget object.

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set()
  list(
    tar_target(y1, 1 + 1),
    tar_target(y2, 1 + 1),
    tar_target(z, y1 + y2)
  )
})
tar_visnetwork()
tar_visnetwork(allow = starts_with("y"))
})
}
```

---

tar_watch *Shiny app to watch the dependency graph.*

Description

Launches a background process with a Shiny app that calls `tar_visnetwork()` every few seconds. To embed this app in other apps, use the Shiny module in `tar_watch_ui()` and `tar_watch_server()`.

Usage

```
tar_watch(
  seconds = 15,
  seconds_min = 1,
  seconds_max = 60,
  seconds_step = 1,
  targets_only = FALSE,
  outdated = TRUE,
  label = NULL,
  level_separation = 150,
  height = "650px",
  background = TRUE,
  browse = TRUE,
  host = getOption("shiny.host", "127.0.0.1"),
  port = getOption("shiny.port", targets::tar_random_port()),
  verbose = TRUE,
  supervise = TRUE
)
```

Arguments

seconds	Numeric of length 1, default number of seconds between refreshes of the graph. Can be changed in the app controls.
seconds_min	Numeric of length 1, lower bound of seconds in the app controls.
seconds_max	Numeric of length 1, upper bound of seconds in the app controls.
seconds_step	Numeric of length 1, step size of seconds in the app controls.
targets_only	Logical, whether to restrict the output to just targets (FALSE) or to also include imported global functions and objects.
outdated	Logical, whether to show colors to distinguish outdated targets from up-to-date targets. (Global functions and objects still show these colors.) Looking for outdated targets takes a lot of time for large pipelines with lots of branches, and setting outdated to FALSE is a nice way to speed up the graph if you only want to see dependency relationships and build progress.
label	Label argument to tar_visnetwork().
level_separation	Numeric of length 1, levelSeparation argument of visNetwork::visHierarchicalLayout(). Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If level_separation is NULL, the levelSeparation argument of visHierarchicalLayout() defaults to 150.
height	Character of length 1, height of the visNetwork widget and branches table.
background	Logical, whether to run the app in a background process so you can still use the R console while the app is running.
browse	Whether to open the app in a browser when the app is ready. Only relevant if background is TRUE.

host	Character of length 1, IPv4 address to listen on. Only relevant if background is TRUE.
port	Positive integer of length 1, TCP port to listen on. Only relevant if background is TRUE.
verbose	whether to print a spinner and informative messages. Only relevant if background is TRUE.
supervise	Whether to register the process with a supervisor. If TRUE, the supervisor will ensure that the process is killed when the R process exits.

Details

The controls of the app are in the left panel. The seconds control is the number of seconds between refreshes of the graph, and the other settings match the arguments of tar_visnetwork().

Value

A handle to callr::r_bg() background process running the app.

Examples

```
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  sleep_run <- function(...) {
    Sys.sleep(10)
  }
  list(
    tar_target(settings, sleep_run()),
    tar_target(data1, sleep_run(settings)),
    tar_target(data2, sleep_run(settings))
  )
}, ask = FALSE)
# Launch the app in a background process.
tar_watch(seconds = 10, outdated = FALSE, targets_only = TRUE)
# Run the pipeline.
tar_make()
})
}
```

---

tar_watch_server *Shiny module server for tar_watch()*

Description

Use tar_watch_ui() and tar_watch_server() to include tar_watch() as a Shiny module in an app.

Usage

```
tar_watch_server(id, height = "650px")
```

Arguments

id	An ID string that corresponds with the ID used to call the module's UI function.
height	Character of length 1, height of the visNetwork widget and branches table.

Value

A Shiny module server.

---

tar_watch_ui *Shiny module UI for tar_watch()*

Description

Use tar_watch_ui() and tar_watch_server() to include tar_watch() as a Shiny module in an app.

Usage

```
tar_watch_ui(
  id,
  label = "tar_watch_label",
  seconds = 5,
  seconds_min = 1,
  seconds_max = 60,
  seconds_step = 1,
  targets_only = FALSE,
  outdated = TRUE,
  label_tar_visnetwork = NULL,
  level_separation = 150,
  height = "650px"
)
```

Arguments

id	An ID string that corresponds with the ID used to call the module's UI function.
label	Label for the module.
seconds	Numeric of length 1, default number of seconds between refreshes of the graph. Can be changed in the app controls.
seconds_min	Numeric of length 1, lower bound of seconds in the app controls.
seconds_max	Numeric of length 1, upper bound of seconds in the app controls.
seconds_step	Numeric of length 1, step size of seconds in the app controls.

`targets_only` Logical, whether to restrict the output to just targets (`FALSE`) or to also include imported global functions and objects.

`outdated` Logical, whether to show colors to distinguish outdated targets from up-to-date targets. (Global functions and objects still show these colors.) Looking for outdated targets takes a lot of time for large pipelines with lots of branches, and setting `outdated` to `FALSE` is a nice way to speed up the graph if you only want to see dependency relationships and build progress.

`label_tar_visnetwork`
Character vector, `label` argument to `tar_visnetwork()`.

`level_separation`
Numeric of length 1, `levelSeparation` argument of `visNetwork::visHierarchicalLayout()`. Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If `level_separation` is `NULL`, the `levelSeparation` argument of `visHierarchicalLayout()` defaults to `150`.

`height` Character of length 1, height of the `visNetwork` widget and branches table.

Value

A Shiny module UI.

`tar_workspace` *Load a saved workspace and seed for debugging.*

Description

Load the packages, workspace, and random number generator seed of target attempted with a workspace file.

Usage

```
tar_workspace(name, envir = parent.frame(), packages = TRUE, source = TRUE)
```

Arguments

`name` Symbol, name of the target whose workspace to read.

`envir` Environment in which to put the objects.

`packages` Logical, whether to load the required packages of the target.

`source` Logical, whether to run _targets.R to load user-defined global object dependencies into `envir`. If `TRUE`, then `envir` should either be the global environment or inherit from the global environment.

Details

If you set `error = "workspace"` in `tar_option_set()` or `tar_target()`, then if that target throws an error in `tar_make()`, it will save its workspace to an RDS file in _targets/workspaces/. Workspaces also get saved for targets supplied to the `workspace` argument of `tar_option_set()`, even individual branches. The workspace is a compact reference that allows `tar_workspace()` to load the target's dependencies and random number generator seed as long as the data objects are still in the data store (usually files in _targets/objects/). When you are done debugging, you can remove the workspace files using `tar_destroy(destroy = "workspaces")`.

Value

This function returns `NULL`, but it does load the target's required packages, as well as multiple objects into the environment (`envir` argument) in order to replicate the workspace where the error happened. These objects include the global objects at the time `tar_make()` was called and the dependency targets. The random number generator seed for the target is also assigned with `set.seed()`

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tmp <- sample(1)
tar_script({
  tar_option_set(error = "workspace")
  list(
    tar_target(x, "loaded"),
    tar_target(y, stop(x))
  )
}, ask = FALSE)
# The following code throws an error for demonstration purposes.
try(tar_make())
exists("x") # Should be FALSE.
tail(.Random.seed) # for comparison to the RNG state after tar_workspace(y)
tar_workspace(y)
exists("x") # Should be TRUE.
print(x) # "loaded"
# Should be different: tar_workspace() runs set.seed(tar_meta(y, seed)$seed)
tail(.Random.seed)
})
}
```

---

tar_workspaces *List saved target workspaces.*

Description

List target workspaces currently saved to _targets/workspaces/. See `tar_workspace()` for more information.

Usage

```
tar_workspaces(names = NULL)
```

Arguments

names	Optional tidyselect selector to return a tactical subset of workspace names. If NULL, all names are selected.

Value

Character vector of available workspaces to load with tar_workspace().

Examples

```
if (identical(Sys.getenv("TAR_LONG_EXAMPLES"), "true")) {
tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
  tar_option_set(error = "workspace")
  list(
    tar_target(x, "value"),
    tar_target(y, x)
  )
}, ask = FALSE)
tar_make()
tar_workspaces()
tar_workspaces(contains("x"))
})
}
```

Index