Package ‘targets’

August 20, 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  0.7.0

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), tools, utils, vctrs (>= 0.2.4), withr
         (>= 2.1.2), yaml (>= 2.2.1)

Suggests  arrow (>= 3.0.0), aws.s3 (>= 0.3.21), bs4Dash (>= 0.5.0),
           clustermq (>= 0.8.95.1), curl (>= 4.3), DT (>= 0.14), 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 (>= 1.32),
           markdown (>= 1.1), rmarkdown (>= 2.4), pingr (>= 2.0.1),
           pkgload (>= 1.1.0), qsl (>= 0.24.1), rstudioapi (>= 0.11), shiny
           (>= 1.5.0), shinybusy (>= 0.2.2), 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  2021-08-19 22:00:02 UTC

R topics documented:

- targets-package .................................................. 4
- tar_assert ......................................................... 5
- tar_branches ....................................................... 7
- tar_branch_index .................................................. 8
- tar_branch_names ............................................... 9
- tar_branch_names_raw .......................................... 10
- tar_built ......................................................... 11
- tar_cancel ......................................................... 12
- tar_canceled ...................................................... 13
- tar_condition ..................................................... 14
- tar_config_get ................................................... 14
- tar_config_set ................................................... 15
- tar_cue ............................................................ 17
- tar_delete ........................................................ 19
- tar_deps .......................................................... 20
- tar_deps_raw ...................................................... 21
- tar_destroy ......................................................... 22
- tar_dir ............................................................ 23
- tar_edit ........................................................... 24
- tar_engine_knitr .................................................. 24
- tar_envir .......................................................... 26
- tar_envvars ....................................................... 27
- tar_errored ........................................................ 28
- tar_exist_meta .................................................... 29
- tar_exist_objects ................................................. 30
- tar_exist_process ............................................... 31
- tar_exist_progress .............................................. 31
- tar_exist_script ............................................... 32
R topics documented:

tar_github_actions .................................................. 33
tar_glimpse ............................................................. 34
tar_group ................................................................. 36
tar_helper ................................................................. 38
tar_helper_raw .......................................................... 39
tar_interactive .......................................................... 40
tar_invalidate ............................................................ 41
tar_language .............................................................. 42
tar_load ..................................................................... 43
tar_loadGlobals .......................................................... 44
tar_load_raw .............................................................. 45
tar_make .................................................................... 47
tar_make_clusterMQ ....................................................... 49
tar_make_future .......................................................... 51
tar_manifest ............................................................... 54
tar_meta ................................................................. 56
tar_name ................................................................. 58
tar_network ............................................................... 59
tar_newer ................................................................. 62
tar_noninteractive ...................................................... 63
tar_objects ............................................................... 64
tar_older ................................................................. 65
tar_option_get ............................................................ 66
tar_option_reset .......................................................... 67
tar_option_set ............................................................ 68
tar_outdated ............................................................. 72
tar_path .................................................................. 74
tar_pattern ............................................................... 75
tar_pid ................................................................. 77
tar_poll ................................................................. 78
tar_process .............................................................. 79
tar_progress ............................................................. 81
tar_progress_branches .................................................. 82
tar_progress_summary ................................................... 83
tar_prune ................................................................. 85
tar_read ................................................................. 86
tar_read_raw ............................................................. 87
tar_renv ................................................................. 89
tar_resources ........................................................... 91
tar_resources_aws ....................................................... 93
tar_resources_clusterMQ ................................................ 94
tar_resources_feather ................................................... 95
tar_resources_fst ........................................................ 96
tar_resources_future .................................................... 97
tar_resources_parquet .................................................. 98
tar_resources_qs ........................................................ 99
tar_resources_url ........................................................ 100
tar_script ............................................................... 101
Library targets-package

Index

Index

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).

See Also

Other help: use_targets()
**tar_assert**

**Assertions**

**Description**

These functions assert the correctness of user inputs and generate custom error conditions as needed. Useful for writing packages built on top of `targets`.

**Usage**

- `tar_assert_chr(x, msg = NULL)`
- `tar_assert_dbl(x, msg = NULL)`
- `tar_assert_df(x, msg = NULL)`
- `tar_assert_equal_lengths(x, msg = NULL)`
- `tar_assert_envir(x, msg = NULL)`
- `tar_assert_expr(x, msg = NULL)`
- `tar_assert_flag(x, choices, msg = NULL)`
- `tar_assert_function(x, msg = NULL)`
- `tar_assert_ge(x, threshold, msg = NULL)`
- `tar_assert_identical(x, y, msg = NULL)`
- `tar_assert_in(x, choices, msg = NULL)`
- `tar_assert_not_dirs(x, msg = NULL)`
- `tar_assert_not_dir(x, msg = NULL)`
- `tar_assert_not_in(x, choices, msg = NULL)`
- `tar_assert_inherits(x, class, msg = NULL)`
- `tar_assert_int(x, msg = NULL)`
- `tar_assert_internet(msg = NULL)`
- `tar_assert_lang(x, msg = NULL)`
- `tar_assert_le(x, threshold, msg = NULL)`
tar_assert_list(x, msg = NULL)
tar_assert_lgl(x, msg = NULL)
tar_assert_name(x)
tar_assert_names(x, msg = NULL)
tar_assert_nonempty(x, msg = NULL)
tar_assert_not_expr(x, msg = NULL)
tar_assert_nzchar(x, msg = NULL)
tar_assert_package(package)
tar_assert_path(path, msg = NULL)
tar_assert_match(x, pattern, msg = NULL)
tar_assert_nonmissing(x, msg = NULL)
tar_assert_positive(x, msg = NULL)
tar_assert_scalar(x, msg = NULL)
tar_assert_target(x, msg = NULL)
tar_assert_target_list(x)
tar_assert_true(x, msg = NULL)
tar_assert_unique(x, msg = NULL)
tar_assert_unique_targets(x)

Arguments

x R object, input to be validated. The kind of object depends on the specific assertion function called.
msg Character of length 1, a message to be printed to the console if x is invalid.
choices Character vector of choices of x for certain assertions.
threshold Numeric of length 1, lower/upper bound for assertions like tar_assert_le()/tar_assert_ge().
y R object, value to compare against x.
class Character vector of expected class names.
package Character of length 1, name of an R package.
### tar_branches

<table>
<thead>
<tr>
<th>path</th>
<th>Character, file path.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>Character of length 1, a grep pattern for certain assertions.</td>
</tr>
</tbody>
</table>

**See Also**

Other utilities to extend targets: `tar_condition`, `tar_dir()`, `tar_language`, `tar_test()`

**Examples**

```R
  tar_assert_chr("123")
  try(tar_assert_chr(123))
```

---

**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**

```R
  tar_branches(name, pattern, store = targets::tar_config_get("store"))
```

**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.
- **store**: Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `.targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**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.)

See Also

Other branching: `tar_branch_index()`, `tar_branch_names_raw()`, `tar_branch_names()`, `tar_pattern()`

Examples

```r
if (identical(Sys.getenv("TAR_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_branch_index**  
**Integer branch indexes**

Description

Get the integer indexes of individual branch names within their corresponding dynamic branching targets.

Usage

`tar_branch_index(names, store = targets::tar_config_get("store"))`

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>names</td>
<td>Character vector of branch names</td>
</tr>
<tr>
<td>store</td>
<td>Character of length 1, path to the targets data store. Defaults to <code>tar_config_get(&quot;store&quot;)</code>, which in turn defaults to <code>.targets/</code>. When you set this argument, the value of <code>tar_config_get(&quot;store&quot;)</code> is temporarily changed for the current function call. See <code>tar_config_get()</code> and <code>tar_config_set()</code> for details about how to set the data store path persistently for a project.</td>
</tr>
</tbody>
</table>

Value

A named integer vector of branch indexes.
See Also

Other branching: `tar_branch_names_raw()`, `tar_branch_names()`, `tar_branches()`, `tar_pattern()`

Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({
    # tar_dir() runs code from a temporary directory.
    tar_script({
      list(
        tar_target(x, seq_len(4)),
        tar_target(y, 2 * x, pattern = map(x)),
        tar_target(z, y, pattern = map(y))
      ),
    }, ask = FALSE)
    tar_make()
    names <- c(
      tar_meta(y, children)$children[[1]][c(2, 3)],
      tar_meta(z, children)$children[[1]][2]
    )
    names
    tar_branch_index(names) # c(2, 3, 2)
  })
}
```

```
tar_branch_names  Branch names
```

Description

Get the branch names of a dynamic branching target using numeric indexes.

Usage

```r
tar_branch_names(name, index, store = targets::tar_config_get("store"))
```

Arguments

- `name` Symbol, name of the dynamic branching target (pattern).
- `index` Integer vector of branch indexes.
- `store` Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `.targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

A character vector of branch names.
See Also

Other branching: \texttt{tar\_branch\_index()}, \texttt{tar\_branch\_names\_raw()}, \texttt{tar\_branches()}, \texttt{tar\_pattern()}

Examples

\begin{verbatim}
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({ # tar\_dir() runs code from a temporary directory.
    tar_script({
      list(
        tar_target(x, seq_len(4)),
        tar_target(y, 2 * x, pattern = map(x)),
        tar_target(z, y, pattern = map(y))
      ),
      ask = FALSE
    })
    tar_make()
    tar_branch_names(z, c(2, 3))
  })
}
\end{verbatim}

Description

Get the branch names of a dynamic branching target using numeric indexes. Same as \texttt{tar\_branch\_names()} except name is a character of length 1.

Usage

\begin{verbatim}
tar_branch_names_raw(name, index, store = targets::tar_config_get("store"))
\end{verbatim}

Arguments

\begin{itemize}
  \item \texttt{name} Character of length 1, name of the dynamic branching target (pattern).
  \item \texttt{index} Integer vector of branch indexes.
  \item \texttt{store} Character of length 1, path to the targets data store. Defaults to \texttt{tar_config_get("store")}, which in turn defaults to \_targets/\. When you set this argument, the value of \texttt{tar_config_get("store")} is temporarily changed for the current function call. See \texttt{tar_config\_get()} and \texttt{tar_config\_set()} for details about how to set the data store path persistently for a project.
\end{itemize}

Value

A character vector of branch names.

See Also

Other branching: \texttt{tar\_branch\_index()}, \texttt{tar\_branch\_names()}, \texttt{tar\_branches()}, \texttt{tar\_pattern()}

---

\texttt{tar\_branch\_names\_raw} \hspace{1cm} Branch names (raw version)
Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir(# tar_dir() runs code from a temporary directory.
  tar_script({
    list(
      tar_target(w, 1),
      tar_target(x, seq_len(4)),
      tar_target(y, 2 * x, pattern = map(x)),
      tar_target(z, y, pattern = map(y))
    ), ask = FALSE)
  tar_make()
  tar_branch_names_raw("z", c(2, 3))
})
}
```

### tar_built

List built targets.

#### Description

List targets whose progress is "built".

#### Usage

```r
tar_built(names = NULL, store = targets::tar_config_get("store"))
```

#### Arguments

- **names**: Optional, names of the targets. If supplied, the function restricts its output to these targets. You can supply symbols or tidyselect helpers like `all_of()` and `starts_with()`.
- **store**: Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

#### Value

A character vector of built targets.

#### See Also

Other progress: `tar_canceled()`, `tar_errored()`, `tar_poll()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_progress()`, `tar_skipped()`, `tar_started()`, `tar_watch_server()`, `tar_watch_ui()`, `tar_watch()`
Examples

```r
if (identical(Sys.getenv("TAR_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_built()
  tar_built(starts_with("y_")) # see also all_of()
}
}
```

---

**Description**

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

**Usage**

```r
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.

**See Also**

Other utilities: `tar_envir()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_seed()`

**Examples**

```r
if (identical(Sys.getenv("TAR_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.
  })
}
```
List canceled targets.

Description

List targets whose progress is "canceled".

Usage

tar_canceled(names = NULL, store = targets::tar_config_get("store"))

Arguments

names  Optional, names of the targets. If supplied, the function restricts its output to these targets. You can supply symbols or tidyselect helpers like all_of() and starts_with().

store  Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Value

A character vector of canceled targets.

See Also

Other progress: tar_built(), tar_errored(), tar_poll(), tar_progress_branches(), tar_progress_summary(), tar_progress(), tar_skipped(), tar_started(), tar_watch_server(), tar_watch_ui(), tar_watch()

Examples

if (identical(Sys.getenv("TAR_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)
    }, ask = FALSE)
    tar_make()
    tar_canceled()
    tar_canceled(starts_with("y_"))  # see also all_of()
  })
}
**tar_condition**

*Conditions*

**Description**

These functions throw custom targets-specific error conditions. Useful for error handling in packages built on top of targets.

**Usage**

- `tar_message_run(...)`
- `tar_throw_file(...)`
- `tar_throw_run(...)`
- `tar_throw_validate(...)`
- `tar_warn_deprecate(...)`
- `tar_warn_run(...)`
- `tar_warn_validate(...)`

**Arguments**

```
... zero or more objects which can be coerced to character (and which are pasted together with no separator) or a single condition object.
```

**See Also**

Other utilities to extend targets: `tar_assert, tar_dir(), tar_language, tar_test()`

**Examples**

```r
try(tar_throw_validate("something is not valid"))
```

---

**tar_config_get**

*Get configuration settings from _targets.yaml.*

**Description**

Read the custom settings in the optional _targets.yaml configuration file at the current project.

**Usage**

```
tar_config_get(name, config = "_targets.yaml")
```
**Arguments**

**name**
Character of length 1, name of the specific configuration setting to retrieve. If name is "config", then instead of retrieving a specific setting, then `tar_config_get()` will return the path to the YAML file where those settings are stored for the current R session (default: _targets.yaml).

**config**
Character of length 1, path to the YAML file with all the configuration settings (default: _targets.yaml).

**Details**

Each project can have an optional YAML configuration file (default: _targets.yaml) with settings specific to a given project. You can write it by hand or modify it with `tar_config_set()`. The currently supported configuration settings are documented as the arguments of `tar_config_set()`.

**Value**

The value of the configuration setting from the YAML configuration file (default: _targets.yaml) or the default value if the setting is not available. The data type of the return value depends on your choice of name.

**See Also**

Other configuration: `tar_config_set()`, `tar_envvars()`, `tar_option_get()`, `tar_option_reset()`, `tar_option_set()`

**Examples**

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({ # tar_dir() runs code from a temporary directory.
    tar_script(list(tar_target(x, 1 + 1)))
    tar_config_get("store") # ",_targets"
    store_path <- tempfile()
    tar_config_set(store = store_path)
    tar_config_get("store") # Shows a temp file.
    tar_make() # Writes to the custom data store identified in _targets.yaml.
    tar_read(x) # tar_read() knows about _targets.yaml too.
    file.exists("_targets") # FALSE
    file.exists(store_path) # TRUE
  })
}
```

---

**Description**

`tar_config_set()` writes special custom settings to an optional project-level YAML configuration file (default: _targets.yaml). Most of these settings are default arguments shared across multiple functions called outside _targets.R.
**Usage**

```r
tar_config_set(
    config = "_targets.yaml",
    reporter_make = NULL,
    reporter_outdated = NULL,
    store = NULL,
    shortcut = NULL,
    script = NULL,
    workers = NULL
)
```

**Arguments**

- **config**: Character of length 1, path to the YAML file with all the configuration settings (default: `_targets.yaml`).
- **reporter_make**: Character of length 1, reporter argument to `tar_make()` and related functions that run the pipeline.
- **reporter_outdated**: Character of length 1, reporter argument to `tar_outdated()` and related functions that do not run the pipeline.
- **store**: Character of length 1, path to the data store of the pipeline. If NULL, the store setting is left unchanged in the YAML configuration file (default: `_targets.yaml`). Usually, the data store lives at `_targets`. Set `store` to a custom directory to specify a path other than `_targets/`. The path need not exist before the pipeline begins, and it need not end with "/_targets"", but it must be writeable. For optimal performance, choose a storage location with fast read/write access.
- **shortcut**: Logical of length 1, default shortcut argument to `tar_make()` and related functions.
- **script**: Character of length 1, path to the target script file that defines the pipeline (`_targets.R` by default). This path should be either an absolute path or a path relative to the project root where you will call `tar_make()` and other functions. When `tar_make()` and friends run the script from the current working directory.
- **workers**: Positive numeric of length 1, workers argument of `tar_make_clustermq()` and related functions that run the pipeline with parallel computing among targets.

**Details**

Each project can have an optional YAML configuration file (default: `_targets.yaml` at the project root) with settings specific to a given project. You can either write it by hand or modify it with `tar_config_set()`, but `tar_config_set()` is recommended because it has guardrails to validate user input. The currently supported configuration settings are documented as the arguments of `tar_config_set()`.

`tar_config_set()` always writes a YAML file with a full set of configuration settings even when no arguments are supplied. To reset options completely, simply call `tar_config_set(config = "_targets.yaml")` and remove `_targets.yaml` if it exists.
**Value**

NULL (invisibly)

**See Also**

Other configuration: `tar_config_get()`, `tar_envvars()`, `tar_option_get()`, `tar_option_reset()`, `tar_option_set()`

**Examples**

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({ 
    tar_script(list(tar_target(x, 1 + 1))))
  tar_config_get("store") # NULL (data store defaults to "_targets/")
  store_path <- tempfile()
  tar_config_set(store = store_path)
  tar_config_get("store") # Shows a temp file.
  tar_make() # Writes to the custom data store identified in _targets.yaml.
  tar_read(x) # Can read from _targets.yaml too.
  file.exists(_targets) # FALSE
  file.exists(store_path) # TRUE
})
```

**Description**

Declare the rules that cue a target.

**Usage**

```r
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.

Target invalidation rules

targets uses internal metadata and special cues to decide whether a target is up to date (can skip) or is outdated/invalidated (needs to rerun). By default, targets moves through the following list of cues and declares a target outdated if at least one is cue activated.

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.

The user can suppress many of the above cues using the `tar_cue()` function, which creates the cue argument of `tar_target()`. Cues objects also constitute more nuanced target invalidation rules. The tarchetypes package has many such examples, including `tar_age()`, `tar_download()`, `tar_cue_age()`, `tar_cue_force()`, and `tar_cue_skip()`.

Dependency-based invalidation and user-defined functions

If the cue of a target has `depend = TRUE` (default) then the target is marked invalidated/invalidated when its upstream dependencies change. A target’s dependencies include upstream targets, user-defined functions, and other global objects populated in the target script file (default: _targets.R). To determine if a given dependency changed since the last run of the pipeline, targets computes hashes. The hash of a target is computed on its files in storage (usually a file in _targets/objects/). The hash of a non-function global object dependency is computed directly on its in-memory data. User-defined functions are hashed in the following way:

1. Deparse the function with `targets:::tar_deparse_safe()`. This function computes a string representation of the function body and arguments. This string representation is invariant to changes in comments and whitespace, which means 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. Using static code analysis (i.e. `tar_deps()`) which is based on `codetools:::findGlobals()` identify any user-defined functions and global objects that the current function depends on. Append the hashes of those dependencies to the string representation of the current function.

4. Compute the hash of the final string representation using `targets:::digest_chr64()`.

Above, (3) is important because user-defined functions have dependencies of their own, such as other user-defined functions and other global objects. (3) ensures that a change to a function’s dependencies invalidates the function itself, which in turn invalidates any calling functions and any targets downstream with the depend cue turned on.

See Also

Other targets: `tar_target_raw()`, `tar_target()`

Examples

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

---

**tar_delete**

*Delete locally stored target return values.*

**Description**

Delete the return values of targets in `_targets/objects/` but keep the records in `_targets/meta/meta`. Dynamic files and cloud data (e.g. `format = "file"` and `format = "aws_parquet"`) are not deleted.

**Usage**

```r
tar_delete(names, store = targets:::tar_config_get("store"))
```

**Arguments**

- `names` Names of the targets to remove from `_targets/objects/`. You can supply symbols or tidyselect helpers like `all_of()` and `starts_with()`.
- `store` Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Details**

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

Other clean: `tar_destroy()`, `tar_invalidate()`, `tar_prune()`

Examples

```r
if (identical(Sys.getenv("TAR_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.

**See Also**

Other inspect: `tar_deps_raw()`, `tar_glimpse()`, `tar_manifest()`, `tar_network()`, `tar_outdated()`, `tar_sitrep()`, `tar_validate()`, `tar_visnetwork()`
**Examples**

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

---

**Description**

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

**Usage**

```r
tar_deps_raw(expr)
```

**Arguments**

- `expr` An R expression object or function.

**Value**

Character vector of the dependencies of a function or expression.

**See Also**

Other inspect: `tar_deps()`, `tar_glimpse()`, `tar_manifest()`, `tar_network()`, `tar_outdated()`, `tar_sitrep()`, `tar_validate()`, `tar_visnetwork()`

**Examples**

```r
tar_deps_raw(quote(x <- y + z))
tar_deps_raw(quote(
  quote({
    x <- 1
    x + a
  })
))
tar_deps_raw(function(a = b) map_dfr(data, ~do_row(.x)))
```
tar_destroy  
Destroy all or part of the data store.

Description
Destroy all or part of the data store written by `tar_make()` and similar functions.

Usage
```r
tar_destroy(
  destroy = c("all", "meta", "process", "progress", "objects", "scratch", "workspaces"),
  ask = NULL,
  store = targets::tar_config_get("store")
)
```

Arguments

- **destroy**: Character of length 1, what to destroy. Choices:
  - "all": destroy the entire data store (default: `_targets/`)
  - "meta": just delete the metadata file at `meta/meta` in the data store, which invalidates all the targets but keeps the data.
  - "process": just delete the progress data file at `meta/process` in the data store, which resets the metadata of the main process.
  - "progress": just delete the progress data file at `meta/progress` in the data store, which resets the progress tracking info.
  - "objects": delete all the target return values in `objects/` in the data store 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 `workspaces/` in the data store with the saved workspaces of targets. See `tar_workspace()` for details.

- **ask**: Logical of length 1, whether to pause with a menu prompt before deleting files. To disable this menu, set the `TAR_ASK` environment variable to "false". `usethis::edit_r_environ()` can help set environment variables.

- **store**: Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

Nothing.
**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

```r
tar_dir(code)
```

## Arguments

- **code**  
  User-defined code.

## Value

Return value of the user-defined code.

## See Also

Other utilities to extend targets: `tar_assert`, `tar_condition`, `tar_language`, `tar_test()`

## Examples

```r
tar_dir(file.create("only_exists_in_tar_dir"))
file.exists("only_exists_in_tar_dir")
```
**tar>Edit**

_Open the target script file for editing._

**Description**

Open the target script file for editing. Requires the usethis package.

**Usage**

```
tar_edit(script = targets::tar_config_get("script"))
```

**Arguments**

- **script**
  
  Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

**Details**

The target script file is an R code file that defines the pipeline. The default path is `_targets.R`, but the default for the current project can be configured with `tar_config_set()`.

**See Also**

Other scripts: `tar_github_actions()`, `tar_helper_raw()`, `tar_helper()`, `tar_renv()`, `tar_script()`

---

**tar_engine_knitr**

_Target Markdown knitr engine_

**Description**

`knitr` language engine that runs `{targets}` code chunks in Target Markdown.

**Usage**

```
tar_engine_knitr(options)
```

**Arguments**

- **options**
  
  A named list of `knitr` chunk options.

**Value**

Character, output generated from `knitr::engine_output()`.
Target Markdown interactive mode

Target Markdown has two modes:

1. Non-interactive mode. This is the default when you run `knitr::knit()` or `rmarkdown::render()`. Here, the code in `{targets}` code chunks gets written to special script files in order to set up a targets pipeline to run later.

2. Interactive mode: here, no scripts are written to set up a pipeline. Rather, the globals or targets in question are run in the current environment and the values are assigned to that environment.

The mode is interactive if `!isTRUE(getOption("knitr.in.progress"))`, is TRUE. The `knitr.in.progress` option is TRUE when you run `knitr::knit()` or `rmarkdown::render()` and NULL if you are running one chunk at a time interactively in an integrated development environment, e.g. the notebook interface in RStudio: https://bookdown.org/yihui/rmarkdown/notebook.html. You can choose the mode with the `tar_interactive` chunk option. (In targets 0.6.0, `tar_interactive` defaults to `interactive()` instead of `!isTRUE(getOption("knitr.in.progress"))`.)

Target Markdown chunk options

Target Markdown introduces the following `knitr` code chunk options. Most other standard `knitr` code chunk options should just work in non-interactive mode. In interactive mode, not all

- `tar_globals`: Logical of length 1, whether to define globals or targets. If TRUE, the chunk code defines functions, objects, and options common to all the targets. If FALSE or NULL (default), then the chunk returns formal targets for the pipeline.

- `tar_interactive`: Logical of length 1, whether to run in interactive mode or non-interactive mode. See the "Target Markdown interactive mode" section of this help file for details.

- `tar_name`: name to use for writing helper script files (e.g. `_targets_r/targets/target_script.R`) and specifying target names if the `tar_simple` chunk option is TRUE. All helper scripts and target names must have unique names, so please do not set this option globally with `knitr::opts_chunk$set()`.

- `tar_script`: Character of length 1, where to write the target script file in non-interactive mode. Most users can skip this option and stick with the default `_targets.R` script path. Helper script files are always written next to the target script in a folder with an "_r" suffix. The `tar_script` path must either be absolute or be relative to the project root (where you call `tar_make()` or similar). If not specified, the target script path defaults to `tar_config_get("script")` (default: `_targets.R`; helpers default: `_targets_r/`). When you run `tar_make()` etc. with a non-default target script, you must select the correct target script file either with the script argument or with `tar_config_set(script = ...)`. The function will source() the script file from the current working directory (i.e. with `chdir = FALSE` in source()).

- `tar_simple`: Logical of length 1. Set to TRUE to define a single target with a simplified interface. In code chunks with `tar_simple` equal to TRUE, the chunk label (or the `tar_name` chunk option if you set it) becomes the name, and the chunk code becomes the command. In other words, a code chunk with label targetname and command mycommand() automatically gets converted to `tar_target(name = targetname, command = mycommand())`. All other arguments of `tar_target()` remain at their default values (configurable with `tar_option_set()` in a `tarGlobals = TRUE` chunk).
See Also

https://books.ropensci.org/targets/markdown.html

Other Target Markdown: tar_interactive(), tar_noninteractive(), tar_toggle()

Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  # Register the engine.
  if (requireNamespace("knitr", quietly = TRUE)) {
    knitr::knit_engines$set(targets = targets::tar_engine_knitr)
  }
  # Then, {targets} code chunks in a knitr report will run
  # as described at https://books.ropensci.org/targets/markdown.html.
}
```

---

`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()`).

See Also

Other utilities: `tar_cancel()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_seed()`
Examples

```r
tar_envir()
tar_envir(default = new.env(parent = emptyenv()))
if (identical(Sys.getenv("TAR_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)
})
}
```

Description

You can customize the behavior of `targets` with special environment variables. The sections in this help file describe each environment variable, and the `tar_envvars()` function lists their current values.

Usage

```r
tar_envvars(unset = "")
```

Arguments

- **unset**: Character of length 1, value to return for any environment variable that is not set.

Details

If you modify environment variables, please set them in project-level `.Renviron` file so you do not lose your configuration when you restart your R session. Modify the project-level `.Renviron` file with `usethis::edit_r_environ(scope = "project")`. Restart your R session after you are done editing.

Value

A data frame with one row per environment variable and columns with the name and current value of each. An unset environment variable will have a value of "" by default. (Customize with the `unset` argument).

TAR_ASK

The `TAR_ASK` environment variable accepts values "true" and "false". If `TAR_ASK` is not set, or if it is set to "true", then `targets` asks permission in a menu before overwriting certain files, such as the target script file (default: `_targets.R`) in `tar_script()`. If `TAR_ASK` is "false", then `targets` overwrites the old files with the new ones without asking. Once you are comfortable with `tar_script()`, `tar_github_actions()`, and similar functions, you can safely set `TAR_ASK` to "false" in either a project-level or user-level `.Renviron` file.
TAR_WARN

The TAR_WARN environment variable accepts values "true" and "false". If TAR_WARN is not set, or if it is set to "true", then targets throws warnings in certain edge cases, such as target/global name conflicts and dangerous use of devtools::load_all(). If TAR_WARN is "false", then targets does not throw warnings in these cases. These warnings can detect potentially serious issues with your pipeline, so please do not set TAR_WARN unless your use case absolutely requires it.

See Also

Other configuration: tar_config_get(), tar_config_set(), tar_option_get(), tar_option_reset(), tar_option_set()

Examples

tar_envvars()

tar_errored()  

| tar_errored | List errored targets. |

Description

List targets whose progress is "errored".

Usage

tar_errored(names = NULL, store = targets::tar_config("store"))

Arguments

names  

Optional, names of the targets. If supplied, the function restricts its output to these targets. You can supply symbols or tidyselect helpers like all_of() and starts_with().

store  

Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Value

A character vector of errored targets.

See Also

Other progress: tar_built(), tar_canceled(), tar_poll(), tar_progress_branches(), tar_progress_summary(), tar_progress(), tar_skipped(), tarStarted(), tar_watch_server(), tar_watch_ui(), tar_watch()
Examples

```r
if (identical(Sys.getenv("TAR_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)
    }), ask = FALSE)
  tar_make()
  tar_error()
  tar_error(starts_with("y_")) # see also all_of()
})
```

---

tar_exist_meta  

Check if target metadata exists.

Description

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

Usage

```r
tar_exist_meta(store = targets::tar_config_get("store"))
```

Arguments

- `store` Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to _targets/. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

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

Value

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

See Also

Other existence: `tar_exist_objects()`, `tar_exist_process()`, `tar_exist_progress()`, `tar_exist_script()`

Examples

```r
tar_exist_meta()
```
**Description**

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

**Usage**

```r
tar_exist_objects(names, store = targets::tar_config_get("store"))
```

**Arguments**

- `names` Character vector of target names.
- `store` Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Details**

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

**See Also**

Other existence: `tar_exist_meta()`, `tar_exist_process()`, `tar_exist_progress()`, `tar_exist_script()`

**Examples**

```r
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(store = targets::tar_config_get("store"))
```

**Arguments**
- **store**: Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Details**
To learn more about local storage in targets, visit [https://books.ropensci.org/targets/files.html#internal-files](https://books.ropensci.org/targets/files.html#internal-files).

**Value**
Logical of length 1, whether the current project’s metadata exists.

**See Also**
Other existence: `tar_exist_meta()`, `tar_exist_objects()`, `tar_exist_progress()`, `tar_exist_script()`

**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(store = targets::tar_config_get("store"))
```
Arguments

store Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

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

Value

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

See Also

Other existence: `tar_exist_meta()`, `tar_exist_objects()`, `tar_exist_process()`, `tar_exist_script()`

Examples

```r
tar_exist_progress()
```

---

**tar_exist_script**

*Check if the target script file exists.*

Description

Check if the target script file exists for the current project. The target script is `_targets.R` by default, but the path can be configured for the current project using `tar_config_set()`.

Usage

```r
tar_exist_script(script = targets::tar_config_get("script"))
```

Arguments

script Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

Value

Logical of length 1, whether the current project’s metadata exists.
**See Also**

Other existence: `tar_exist_meta()`, `tar_exist_objects()`, `tar_exist_process()`, `tar_exist_progress()`

**Examples**

```
tar_exist_script()
```

---

**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](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.

See Also

Other scripts: `tar_edit()`, `tar_helper_raw()`, `tar_helper()`, `tar_renv()`, `tar_script()`

Examples

```r
  tar_glimpse(tempfile())
```

---

**Description**

Analyze the pipeline defined in the target script file (default: `_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**

```r
  tar_glimpse(
    targets_only = TRUE,
    names = NULL,
    shortcut = FALSE,
    allow = NULL,
    exclude = ".Random.seed",
    level_separation = NULL,
    degree_from = 1L,
    degree_to = 1L,
    callr_function = callr::r,
    callr_arguments = targets::callr_args_default(callr_function),
    envir = parent.frame(),
    script = targets::tar_config_get("script"),
    store = targets::tar_config_get("store")
  )
```

**Arguments**

- **targets_only** Logical, whether to restrict the output to just targets (FALSE) or to also include global functions and objects.
- **names** Names of targets. The graph visualization will operate only on these targets (and unless `shortcut` is TRUE, all the targets upstream as well). Selecting a small subgraph using `names` could speed up the load time of the visualization. Unlike `allow`, `names` is invoked before the graph is generated. Set to NULL to
check/build all the targets (default). Otherwise, you can supply symbols or tidyselect helpers like `starts_with()`. Applies to ordinary targets (stem) and whole dynamic branching targets (patterns) but not individual dynamic branches.

**shortcut**  
Logical of length 1, how to interpret the `names` argument. If `shortcut` is `FALSE` (default) then the function checks all targets upstream of `names` as far back as the dependency graph goes. If `TRUE`, then the function only checks the targets in `names` and uses stored metadata for information about upstream dependencies as needed. `shortcut` = `TRUE` increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Also, `shortcut` = `TRUE` only works if you set `names`.

**allow**  
Optional, define the set of allowable vertices in the graph. Unlike `names`, `allow` is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to `NULL` to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols or tidyselect helpers like `starts_with()`.

**exclude**  
Optional, define the set of exclude vertices from the graph. Unlike `names`, `exclude` is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to `NULL` to exclude no vertices. Otherwise, you can supply symbols or tidyselect helpers like `all_of()` and `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.

**degree_from**  
Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_from` controls the number of edges the neighborhood extends upstream.

**degree_to**  
Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_to` controls the number of edges the neighborhood extends downstream.

**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`.

**envir**  
An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc.

The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script.
file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

**script**  
Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

**store**  
Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Value**  
A visNetwork HTML widget object.

**See Also**

Other inspect: `tar_deps_raw()`, `tar_deps()`, `tar_manifest()`, `tar_network()`, `tar_outdated()`, `tar_sitrep()`, `tar_validate()`, `tar_visnetwork()`

**Examples**

```r
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")) # see also all_of()
  })
}
```

---

### `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 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.

See Also

Other utilities: tar_cancel(), tar_envir(), tar_name(), tar_path(), tar_seed()

Examples

if (identical(Sys.getenv("TAR_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)
          )
        )
      )
    })
  })
}
tar_helper

Write a helper R script.

Description
Write a helper R script for a targets pipeline. Could be supporting functions or the target script file (default: _targets.R) 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)
See Also

Other scripts: `tar_edit()`, `tar_github_actions()`, `tar_helper()`, `tar_renv()`, `tar_script()`

Examples

```r
# Without tidy evaluation:
p <- tempfile()
tar_helper(p, x <- 1)
writeLines(readLines(p))

# With tidy evaluation:
y <- 123
tar_helper(p, x <- !!y)
writeLines(readLines(p))
```

---

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 target script file (default: `_targets.R`) itself.

Usage

```r
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)

See Also

Other scripts: `tar_edit()`, `tar_github_actions()`, `tar_helper()`, `tar_renv()`, `tar_script()`
Examples

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

---

tar_interactive

Run if Target Markdown interactive mode is on.

Description

In Target Markdown, run the enclosed code only if interactive mode is activated. Otherwise, do not run the code.

Usage

```r
tar_interactive(code)
```

Arguments

code R code to run if Target Markdown interactive mode is turned on.

Details

Visit <books.ropensci.org/targets/markdown.html> to learn about Target Markdown and interactive mode.

Value

If Target Markdown interactive mode is turned on, the function returns the result of running the code. Otherwise, the function invisibly returns `NULL`.

See Also

Other Target Markdown: `tar_engine_knitr()`, `tar_noninteractive()`, `tar_toggle()`

Examples

```r
tar_interactive(message("In interactive mode."))
```
**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/.

**Usage**

```
  tar_invalidate(names, store = targets::tar_config_get("store"))
```

**Arguments**

- **names**: Names of the targets to remove from the metadata list. You can supply symbols or tidyselect helpers like `all_of()` and `starts_with()`.
- **store**: Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to _targets/. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Details**

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

**See Also**

Other clean: `tar_delete()`, `tar_destroy()`, `tar_prune()`

**Examples**

```r
if (identical(Sys.getenv("TAR_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.
  })
}
```
Description

These functions help with metaprogramming in packages built on top of targets.

Usage

```r
tar_deparse_language(expr)
tar_deparse_safe(expr, collapse = "\n", backtick = TRUE)
tar_tidy_eval(expr, envir, tidy_eval)
tar_tidyselect_eval(names_quosure, choices)
```

Arguments

- `expr` A language object to modify or deparse.
- `collapse` Character of length 1, delimiter in deparsing.
- `backtick` Logical indicating whether symbolic names should be enclosed in backticks if they do not follow the standard syntax.
- `envir` An environment to find objects for tidy evaluation.
- `tidy_eval` Logical of length 1, whether to apply tidy evaluation.
- `names_quosure` An `rlang` quosure with `tidyselect` expressions.
- `choices` A character vector of choices for character elements returned by tidy evaluation.

Details

- `tar_deparse_language()` is a wrapper around `tar_deparse_safe()` which leaves character vectors and NULL objects alone, which helps with subsequent user input validation.
- `tar_deparse_safe()` is a wrapper around `base::deparse()` with a custom set of fast default settings and guardrails to ensure the output always has length 1.
- `tar_tidy_eval()` applies tidy evaluation to a language object and returns another language object.
- `tar_tidyselect_eval()` applies tidyselect selection with some special guardrails around NULL inputs.

See Also

Other utilities to extend targets: `tar_assert`, `tar_condition`, `tar_dir()`, `tar_test()`

Examples

```r
tar_deparse_language(quote(run_model()))
```
**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, store = store),
  envir = parent.frame(),
  store = targets::tar_config_get("store")
)
```

**Arguments**

- `names` Names of the targets to load. You can supply symbols, a character vector, or tidyselect helpers like `all_of()` and `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.
- `store` Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to _targets/. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Value**

Nothing.

**Limited scope**

`tar_read()` and `tar_load()` are only for exploratory analysis and literate programming, and `tar_read_raw()` and `tar_load_raw()` are only for exploratory analysis. `targets` automatically loads the correct dependencies into memory when the pipeline is running, so invoking these functions from inside a target is rarely advisable.
`tar_load_globals`

Load globals for debugging, testing, and prototyping

**Description**

Load user-defined packages, functions, global objects, and settings defined in the target script file (default: `_targets.R`). This function is for debugging, testing, and prototyping only. It is not recommended for use inside a serious pipeline or to report the results of a serious pipeline.

**Usage**

```r
.tar_load_globals(
  envir = parent.frame(),
  script = targets::tar_config_get("script")
)
```

**Arguments**

- `envir` Environment to source the target script (default: `_targets.R`). Defaults to the calling environment.
- `script` Character of length 1, path to the target script file that defines the pipeline (_targets.R by default). This path should be either an absolute path or a path relative to the project root where you will call `tar_make()` and other functions. When `tar_make()` and friends run the script from the current working directory.

**Examples**

```r
if (identical(Sys.getenv("TAR_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")) # see also all_of()
})
}
```
Details

This function first sources the target script file (default: `_targets.R`) to loads all user-defined functions, global objects, and settings into the current R process. Then, it loads all the packages defined in `tar_option_get("packages")` (default: `.packages()`) using `library()` with `lib.loc` defined in `tar_option_get("library")` (default: `NULL`).

Value

NULL (invisibly).

See Also

Other debug: `tar_traceback()`, `tar_workspaces()`, `tar_workspace()`

Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir( # tar_dir() runs code from a temporary directory.
    tar_script({
      tar_option_set(packages = "callr")
      analyze_data <- function(data) {
        summary(data)
      }
      list(
        tar_target(x, 1 + 1),
        tar_target(y, 1 + 1)
      ), ask = FALSE)
    tar_load_globals()
    print(analyze_data)
    print("callr" %in% (.packages()))
  })
}
```

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

```r
tar_load_raw(
  names,
  branches = NULL,
  meta = tar_meta(store = store),
```
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.
- **store**: Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

Nothing.

Limited scope

`tar_read()` and `tar_load()` are only for exploratory analysis and literate programming, and `tar_read_raw()` and `tar_load_raw()` are only for exploratory analysis. `targets` automatically loads the correct dependencies into memory when the pipeline is running, so invoking these functions from inside a target is rarely advisable.

See Also

Other data: `tar_load()`, `tar_meta()`, `tar_objects()`, `tar_pid()`, `tar_process()`, `tar_read_raw()`, `tar_read()`

Examples

```r
if (identical(Sys.getenv("TAR_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
  } else {
    tar_read()
  }
```
Run a pipeline of targets.

**Description**

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

**Usage**

```r
tar_make(
    names = NULL,
    shortcut = targets::tar_config_get("shortcut"),
    reporter = targets::tar_config_get("reporter_make"),
    callr_function = callr::r,
    callr_arguments = targets::callr_args_default(callr_function, reporter),
    envir = parent.frame(),
    script = targets::tar_config_get("script"),
    store = targets::tar_config_get("store")
)
```

**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 `all_of()` and `starts_with()`. Applies to ordinary targets (stem) and whole dynamic branching targets (patterns) by not individual dynamic branches.

- **shortcut**
  
  Logical of length 1, how to interpret the names argument. If shortcut is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. shortcut = TRUE increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. It relies on stored metadata for information about upstream dependencies. shortcut = TRUE only works if you set names.

- **reporter**
  
  Character of length 1, name of the reporter to user. Controls how messages are printed as targets run in the pipeline. Defaults to `tar_config_get("reporter_make")`. 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, started, skipped, build, canceled, or errored). Also show a timestamp ("%H:%M %OS2" strftime() format) of the last time the progress changed and printed to the screen.

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.

envir An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc.

The envir argument of tar_make() and related functions always overrides the current value of tar_option_get("envir") in the current R session just before running the target script file, so whenever you need to set an alternative envir, you should always set it with tar_option_set() from within the target script file. In other words, if you call tar_option_set(envir = envir1) in an interactive session and then tar_make(envir = envir2, callr_function = NULL), then envir2 will be used.

script Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. When you set this argument, the value of tar_config_get("script") is temporarily changed for the current function call. See tar_script(), tar_config_get(), and tar_config_set() for details about the target script file and how to set it persistently for a project.

store Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

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.

See Also

Other pipeline: tar_make_clustermq(), tar_make_future()

Examples

tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script({
tar_option_set()}
```r
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 target script file (default: `_targets.R`). clustermq is not a strict dependency of targets, so you must install clustermq yourself.

Usage

```r
tar_make_clustermq(
  names = NULL,
  shortcut = targets::tar_config_get("shortcut"),
  reporter = targets::tar_config_get("reporter_make"),
  workers = targets::tar_config_get("workers"),
  log_worker = FALSE,
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

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 `all_of()` and `starts_with()`. Applies to ordinary targets (stem) and whole dynamic branching targets (patterns) by not individual dynamic branches. |
**shortcut** Logical of length 1, how to interpret the names argument. If shortcut is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. shortcut = TRUE increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. It relies on stored metadata for information about upstream dependencies. shortcut = TRUE only works if you set names.

**reporter** Character of length 1, name of the reporter to user. Controls how messages are printed as targets run in the pipeline. Defaults to tar_config_get("reporter_make").

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, started, skipped, build, canceled, or errored). Also show a timestamp ("%H:%M %OS2" strftime() format) of the last time the progress changed and printed to the screen.

**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.

**envir** An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc.

The envir argument of tar_make() and related functions always overrides the current value of tar_option_get("envir") in the current R session just before running the target script file, so whenever you need to set an alternative envir, you should always set it with tar_option_set() from within the target script file. In other words, if you call tar_option_set(envir = envir1) in an interactive session and then tar_make(envir = envir2, callr_function = NULL), then envir2 will be used.

**script** Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. When you set this argument, the value of tar_config_get("script") is temporarily changed for the current function call. See tar_script(), tar_config_get(), and tar_config_set() for details about the target script file and how to set it persistently for a project.

**store** Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value
of tar_config_get("store") is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Details

To use with a cluster, you will need to set the global options `clustermq.scheduler` and `clustermq.template` inside the target script file (default: `_targets.R`). To read more about configuring clustermq for your scheduler, visit [https://mschubert.github.io/clustermq/articles/userguide.html#configuration](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.

See Also

Other pipeline: `tar_make_future()`, `tar_make()`

Examples

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

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 target script file (default: `_targets.R`). future is not a strict dependency of targets, so you must install future yourself.
Usage

tar_make_future(
  names = NULL,
  shortcut = targets::tar_config_get("shortcut"),
  reporter = targets::tar_config_get("reporter_make"),
  workers = targets::tar_config_get("workers"),
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)

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 all_of() and starts_with(). Applies to ordinary
targets (stem) and whole dynamic branching targets (patterns) by not individual
dynamic branches.

shortcut Logical of length 1, how to interpret the names argument. If shortcut is FALSE
(default) then the function checks all targets upstream of names as far back as the
dependency graph goes. shortcut = TRUE increases speed if there are a lot of
up-to-date targets, but it assumes all the dependencies are up to date, so please
use with caution. It relies on stored metadata for information about upstream
dependencies. shortcut = TRUE only works if you set names.

reporter Character of length 1, name of the reporter to user. Controls how messages are
printed as targets run in the pipeline. Defaults to tar_config_get("reporter_make").
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, started, skipped, build, canceled, or errored). Also
    show a timestamp ("%H:%M %OS2" strftime() format) of the last time the
    progress changed and printed to the screen.

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.
**envir**  An environment, where to run the target R script (default: _targets.R) if `callr_function` is NULL. Ignored if `callr_function` is anything other than NULL. `callr_function` should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc. The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

**script**  Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to _targets.R. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

**store**  Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to _targets/. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

### 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.

### See Also
Other pipeline: `tar_make_clustermq()`, `tar_make()`

### Examples
```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({ # tar_dir() runs code from a temporary directory.
    tar_script(
      future::plan(future::multisession, workers = 2)
      list(
        tar_target(x, 1 + 1),
        tar_target(y, 1 + 1)
      )
    ), ask = FALSE)
    tar_make_future()
  })
}
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

```r
tar_manifest(
  names = NULL,
  fields = c("name", "command", "pattern"),
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script")
)
```

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 `all_of()` and `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.
  - `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()`).
resources: A list of target-specific resource requirements for \texttt{tar_make_future()}.  
• cue_mode: Cue mode from \texttt{tar_cue()}.  
• cue_depend: Depend cue from \texttt{tar_cue()}.  
• cue_expr: Command cue from \texttt{tar_cue()}.  
• cue_file: File cue from \texttt{tar_cue()}.  
• cue_format: Format cue from \texttt{tar_cue()}.  
• cue_iteration: Iteration cue from \texttt{tar_cue()}.  
• packages: List columns of packages loaded before building the target.  
• library: List column of library paths to load the packages.

callr\_function \hspace{1cm} A function from \texttt{callr} to start a fresh clean R process to do the work. Set to \texttt{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). \texttt{callr\_function} needs to be \texttt{NULL} for interactive debugging, e.g. \texttt{tar\_option\_set(debug = "your\_target")}. However, \texttt{callr\_function} should not be \texttt{NULL} for serious reproducible work.

callr\_arguments \hspace{1cm} A list of arguments to \texttt{callr\_function}.

envir \hspace{1cm} An environment, where to run the target R script (default: \_targets.R) if \texttt{callr\_function} is \texttt{NULL}. Ignored if \texttt{callr\_function} is anything other than \texttt{NULL}. callr\_function should only be \texttt{NULL} for debugging and testing purposes, not for serious runs of a pipeline, etc.

The \texttt{envir} argument of \texttt{tar\_make()} and related functions always overrides the current value of \texttt{tar\_option\_get("envir")} in the current R session just before running the target script file, so whenever you need to set an alternative \texttt{envir}, you should always set it with \texttt{tar\_option\_set()} from within the target script file. In other words, if you call \texttt{tar\_option\_set(envir = envir1)} in an interactive session and then \texttt{tar\_make(envir = envir2, callr\_function = NULL)}, then \texttt{envir2} will be used.

script \hspace{1cm} Character of length 1, path to the target script file. Defaults to \texttt{tar\_config\_get("script")}, which in turn defaults to \_targets.R. When you set this argument, the value of \texttt{tar\_config\_get("script")} is temporarily changed for the current function call. See \texttt{tar\_script()}, \texttt{tar\_config\_get()}, and \texttt{tar\_config\_set()} for details about the target script file and how to set it persistently for a project.

\textbf{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).

\textbf{See Also}

Other inspect: \texttt{tar\_deps\_raw()}, \texttt{tar\_deps()}, \texttt{tar\_glimpse()}, \texttt{tar\_network()}, \texttt{tar\_outdated()}, \texttt{tar\_sitrep()}, \texttt{tar\_validate()}, \texttt{tar\_visnetwork()}
Examples

```r
if (identical(Sys.getenv("TAR_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

```r
tar_meta(
  names = NULL,
  fields = NULL,
  targets_only = FALSE,
  complete_only = FALSE,
  store = targets::tar_config_get("store")
)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>names</td>
<td>Optional, names of the targets. If supplied, tar_meta() only returns metadata on these targets. You can supply symbols or tidyselect helpers like all_of() and starts_with(). If NULL, all names are selected.</td>
</tr>
<tr>
<td>fields</td>
<td>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 or tidyselect helpers like all_of() and starts_with(). The name column is always included first no matter what you select. Choices:</td>
</tr>
<tr>
<td>• name: name of the target or global object.</td>
<td></td>
</tr>
</tbody>
</table>
• type: type of the object: either "function" or "object" for 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: POSIXct object with the time the target’s data in storage was last modified. If the target stores no local file, then the time stamp corresponds to the time the target last ran successfully. Only targets that run commands have time stamps: just non-branching targets and individual dynamic branches. Displayed in the current time zone of the system. If there are multiple outputs for that target, as with file targets, then the maximum time is shown.
• 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).
store Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.
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.

See Also

Other data: `tar_load_raw()`, `tar_load()`, `tar_objects()`, `tar_pid()`, `tar_process()`, `tar_read_raw()`, `tar_read()`

Examples

```r
if (identical(Sys.getenv("TAR_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_")) # see also all_of()
  })
}
```

---

`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.

See Also

Other utilities: tar_cancel(), tar_envir(), tar_group(), tar_path(), tar_seed()

Examples

tar_name()
tar_name(default = "custom_target_name")
if (identical(Sys.getenv("TAR_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 the target script file (default: _targets.R) and return the vertices and edges of the directed acyclic graph of dependency relationships.

Usage

tar_network(
  targets_only = FALSE,
  names = NULL,
  shortcut = FALSE,
  allow = NULL,
  exclude = NULL,
  outdated = TRUE,
  reporter = targets::tar_config_get("reporter_outdated"),
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
Arguments

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

names Names of targets. The graph visualization will operate only on these targets (and unless shortcut is TRUE, all the targets upstream as well). Selecting a small subgraph using names could speed up the load time of the visualization. Unlike allow, names is invoked before the graph is generated. Set to NULL to check/build all the targets (default). Otherwise, you can supply symbols or tidyselect helpers like starts_with(). Applies to ordinary targets (stem) and whole dynamic branching targets (patterns) but not individual dynamic branches.

shortcut Logical of length 1, how to interpret the names argument. If shortcut is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. If TRUE, then the function only checks the targets in names and uses stored metadata for information about upstream dependencies as needed. shortcut = TRUE increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Also, shortcut = TRUE only works if you set names.

allow Optional, define the set of allowable vertices in the graph. Unlike names, allow is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to NULL to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols or tidyselect helpers like starts_with().

exclude Optional, define the set of exclude vertices from the graph. Unlike names, exclude is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to NULL to exclude no vertices. Otherwise, you can supply symbols or tidyselect helpers like all_of() and 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.

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.

envir An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function
should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc.

The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

**script**  
Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to `_targets.R`. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

**store**  
Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

### 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, started, 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.

### See Also

Other inspect: `tar_deps_raw()`, `tar_deps()`, `tar_glimpse()`, `tar_manifest()`, `tar_outdated()`, `tar_sitrep()`, `tar_validate()`, `tar_visnetwork()`

### Examples

```r
if (identical(Sys.getenv("TAR_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)
  })
}
```
List new targets

Description

List all the targets whose last successful run occurred after a certain point in time.

Usage

tar_newer(  
  time,  
  names = NULL,  
  inclusive = FALSE,  
  store = targets::tar_config_get("store")  
)

Arguments

time A POSIXct object of length 1, time threshold. Targets newer than this time stamp are returned. For example, if time = Sys.time -as.difftime(1, units = "weeks") then tar_newer() returns targets newer than one week ago.

names Names of eligible targets. Targets excluded from names will not be returned even if they are newer than the given time. You can supply symbols or tidyselect helpers like all_of() and starts_with(). If NULL, all names are eligible.

inclusive Logical of length 1, whether to include targets built at exactly the time given.

store Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Details

Only applies to targets with recorded time stamps: just non-branching targets and individual dynamic branches. As of targets version 0.6.0, these time stamps are available for these targets regardless of storage format. Earlier versions of targets do not record time stamps for remote storage formats such as "url" or any of the "aws_*" formats.

Value

A character vector of names of old targets with recorded timestamp metadata.

See Also

Other time: tar_older(), tar_timestamp_raw(), tar_timestamp()
tar_noninteractive

Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({
    # tar_dir() runs code from a temporary directory.
    tar_script(
      list(tar_target(x, seq_len(2))), ask = FALSE)
  }, ask = FALSE)
  tar_make()
  # targets newer than 1 week ago
  tar_newer(Sys.time() - as.difftime(1, units = "weeks"))
  # targets newer than 1 week from now
  tar_newer(Sys.time() + as.difftime(1, units = "weeks"))
  # Everything is still up to date.
  tar_make()
  # Invalidate all targets targets newer than 1 week ago
  invalidate_these <- tar_newer(Sys.time() - as.difftime(1, units = "weeks"))
  tar_invalidate(all_of(invalidate_these))
  tar_make()
}
```

tar_noninteractive  Run if Target Markdown interactive mode is not on.

Description

In Target Markdown, run the enclosed code only if interactive mode is not activated. Otherwise, do not run the code.

Usage

```r
tar_noninteractive(code)
```

Arguments

code  R code to run if Target Markdown interactive mode is not turned on.

Details

Visit <books.ropensci.org/targets/markdown.html> to learn about Target Markdown and interactive mode.

Value

If Target Markdown interactive mode is not turned on, the function returns the result of running the code. Otherwise, the function invisibly returns NULL.

See Also

Other Target Markdown: tar_engine_knitr(), tar_interactive(), tar_toggle()
Examples

tar_noninteractive(message("Not in interactive mode."))

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, store = targets::tar_config_get("store"))

Arguments

names Optional tidyselect selector such as all_of() or starts_with() to return a tactical subset of target names. If NULL, all names are selected.

store Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Value

Character vector of targets saved to _targets/objects/.

See Also

Other data: tar_load_raw(), tar_load(), tar_meta(), tar_pid(), tar_process(), tar_read_raw(), tar_read()

Examples

if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({ # tar_dir() runs code from a temporary directory.
    tar_script({
      list(tar_target(x, "value"))
    }, ask = FALSE)
    tar_make()
    tar_objects()
    tar_objects(starts_with("x")) # see also all_of()
  })
}
Description

List all the targets whose last successful run occurred before a certain point in time. Combine with \texttt{tar_invalidate()}, you can use \texttt{tar_older()} to automatically rerun targets at regular intervals. See the examples for a demonstration.

Usage

\begin{verbatim}
Usage

\texttt{tar_older(
  time,
  names = NULL,
  inclusive = FALSE,
  store = targets::tar_config_get("store")
)}
\end{verbatim}

Arguments

- \texttt{time}: A \texttt{POSIXct} object of length 1, time threshold. Targets older than this time stamp are returned. For example, if \texttt{time = Sys.time() - as.difftime(1, units = "weeks")} then \texttt{tar_older()} returns targets older than one week ago.
- \texttt{names}: Names of eligible targets. Targets excluded from \texttt{names} will not be returned even if they are old. You can supply symbols or tidyselect helpers like \texttt{all_of()} and \texttt{starts_with()}. If \texttt{NULL}, all names are eligible.
- \texttt{inclusive}: Logical of length 1, whether to include targets built at exactly the time given.
- \texttt{store}: Character of length 1, path to the \texttt{targets} data store. Defaults to \texttt{targets::tar_config_get("store")}, which in turn defaults to \_targets\/. When you set this argument, the value of \texttt{tar_config_get("store")} is temporarily changed for the current function call. See \texttt{tar_config_get()} and \texttt{tar_config_set()} for details about how to set the data store path persistently for a project.

Details

Only applies to targets with recorded time stamps: just non-branching targets and individual dynamic branches. As of \texttt{targets} version 0.6.0, these time stamps are available for these targets regardless of storage format. Earlier versions of \texttt{targets} do not record time stamps for remote storage formats such as \texttt{"url"} or any of the \texttt{"aws_*"} formats.

Value

A character vector of names of old targets with recorded timestamp metadata.

See Also

Other time: \texttt{tar_newer()}, \texttt{tar_timestamp_raw()}, \texttt{tar_timestamp()}

Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({
    # tar_dir() runs code from a temporary directory.
    tar_script({
      list(tar_target(x, seq_len(2)))
    }, ask = FALSE)
    tar_make()
    # targets older than 1 week ago
    tar_older(Sys.time() - as.difftime(1, units = "weeks"))
    # targets older than 1 week from now
    tar_older(Sys.time() + as.difftime(1, units = "weeks"))
    # Everything is still up to date.
    tar_make()
    # Invalidate all targets targets older than 1 week from now
    # so they run on the next tar_make().
    invalidate_these <- tar_older(Sys.time() + as.difftime(1, units = "weeks"))
    tar_invalidate(all_of(invalidate_these))
    tar_make()
  })
}
```

---

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

```r
tar_option_get(name = NULL, option = NULL)
```

#### Arguments

<table>
<thead>
<tr>
<th>name</th>
<th>Character of length 1, name of an option to get. Must be one of the argument names of <code>tar_option_set()</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>option</td>
<td>Deprecated, use the name argument instead.</td>
</tr>
</tbody>
</table>

#### 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.
See Also

Other configuration: tar_config_get(), tar_config_set(), tar_envvars(), tar_option_reset(), tar_option_set()

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_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**

NULL (invisibly).

**See Also**

Other configuration: **tar_config_get()**, **tar_config_set()**, **tar_envvars()**, **tar_option_get()**, **tar_option_set()**
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_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. Only the non-null arguments are actually set as options. See currently set options with tar_option_get(). To use tar_option_set() effectively, put it in your workflow’s target script file (default: _targets.R) 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,
tar_option_set

retrieval = NULL,
cue = NULL,
debug = NULL,
workspaces = NULL,
workspace_on_error = 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 your target script file (default: `_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 common to all targets in the pipeline. The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.  

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_cluster mq()`, 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. See the "Storage formats" section for a detailed list of possible data storage formats.

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 stops and throws an error. Options:

- "stop": the whole pipeline stops and throws an error.
- "continue": the whole pipeline keeps going.
- "abridge": any currently running targets keep running, but no new targets launch after that. (Visit [https://books.ropensci.org/targets/debugging.html](https://books.ropensci.org/targets/debugging.html) to learn how to debug targets using saved workspaces.)

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 strategy 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()`).

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, 0.001, max(backoff, 0.001 * 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, back-off, visit https://en.wikipedia.org/wiki/Exponential_backoff). Raising backoff is kinder to the CPU etc. but may incur delays in some instances.

resources
Object returned by `tar_resources()` with optional settings for high-performance computing functionality, alternative data storage formats, and other optional capabilities of targets. See `tar_resources()` 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 target names. Could be non-branching targets, whole dynamic branching targets, or individual branch names. `tar_make()` and friends will save workspace files for these targets even if the targets are skipped. Workspace files help with debugging. See `tar_workspace()` for details about workspaces.

workspace_on_error
Logical of length 1, whether to save a workspace file for each target that throws an error. Workspace files help with debugging. See `tar_workspace()` for details about workspaces.

Value
NULL (invisibly).

See Also
Other configuration: `tar_config_get()`, `tar_config_set()`, `tar_envvars()`, `tar_option_get()`, `tar_option_reset()`

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_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,
  shortcut = targets::tar_config_get("shortcut"),
  branches = FALSE,
  targets_only = TRUE,
  reporter = targets::tar_config_get("reporter_outdated"),
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)

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 or tidyselect helpers like all_of() and starts_with(). Applies to ordinary targets (stem) and whole dynamic branching targets (patterns) by not individual dynamic branches.

shortcut Logical of length 1, how to interpret the names argument. If shortcut is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. If TRUE, then the function only checks the targets in
names and uses stored metadata for information about upstream dependencies as needed. shortcut = TRUE increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Also, shortcut = TRUE only works if you set names.

branches Logical of length 1, 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 of length 1, whether to just restrict to targets or to include functions and other global objects from the environment created by running the target script file (default: _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.

envir An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc.

The envir argument of tar_make() and related functions always overrides the current value of tar_option_get("envir") in the current R session just before running the target script file, so whenever you need to set an alternative envir, you should always set it with tar_option_set() from within the target script file. In other words, if you call tar_option_set(envir = envir1) in an interactive session and then tar_make(envir = envir2,callr_function = NULL), then envir2 will be used.

script Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. When you set this argument, the value of tar_config_get("script") is temporarily changed for the current function call. See tar_script(), tar_config_get(), and tar_config_set() for details about the target script file and how to set it persistently for a project.

store Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/s. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.
Details

Requires that you define a pipeline with a target script file (default: `_targets.R`). (See `tar_script()` for details.)

Value

Names of the outdated targets.

See Also

Other inspect: `tar_deps_raw()`, `tar_deps()`, `tar_glimpse()`, `tar_manifest()`, `tar_network()`, `tar_sitrep()`, `tar_validate()`, `tar_visnetwork()`

Examples

```r
if (identical(Sys.getenv("TAR_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)
    ), ask = FALSE)
    tar_outdated()
  })
}
```
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.

store Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

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.

See Also

Other utilities: tar_cancel(), tar_envir(), tar_group(), tar_name(), tar_seed()

Examples

tar_path()

tar_path(your_target)

if (identical(Sys.getenv("TAR_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)
  })
}

Description

Emulate dynamic branching. 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.
- `slice()`: select one or more slices by index, e.g. `slice(x, index = c(3,4))` selects the third and fourth slice or branch of `x`.
- `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.

See Also

Other branching: `tar_branch_index()`, `tar_branch_names_raw()`, `tar_branch_names()`, `tar_branches()`

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
# your target script file (default: `_targets.R`).
list(
```
```r
tar_pid

    tar_target(x, seq_len(2)),
    tar_target(y, head(letters, 2)),
    tar_target(d, c(x, y, z), 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(d, 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**

```r
tar_pid(store = targets::tar_config_get("store"))
```

**Arguments**

- **store** Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.
Details

The main process is the R process invoked by \texttt{tar_make()} or similar. If \texttt{callr\_function} is not \texttt{NULL}, this is an external process, and the pid in the return value will not agree with \texttt{Sys.getpid()} in your current interactive session. The process may or may not be alive. You may want to check it with \texttt{ps::ps\_is\_running(ps::ps\_handle(targets::tar\_pid()))} before running another call to \texttt{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.

See Also

Other data: \texttt{tar\_load\_raw()}, \texttt{tar\_load()}, \texttt{tar\_meta()}, \texttt{tar\_objects()}, \texttt{tar\_process()}, \texttt{tar\_read\_raw()}, \texttt{tar\_read()}

Examples

if (identical(Sys.getenv("TAR\_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.
  })
}

\section*{Description}

Print the information in \texttt{tar\_progress\_summary()} at regular intervals.

Usage

\begin{verbatim}
tar_poll(
  interval = 1,
  timeout = Inf,
  fields = c("skipped", "started", "built", "errored", "canceled", "since"),
  store = targets::tar\_config\_get("store")
)
\end{verbatim}
Arguments

- **interval**: Number of seconds to wait between iterations of polling progress.
- **timeout**: How many seconds to run before exiting.
- **fields**: Optional, names of progress data columns to read. Set to NULL to read all fields.
- **store**: Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to _targets_. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

See Also

Other progress: `tar_built()`, `tar_canceled()`, `tar_errored()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_progress()`, `tar_skipped()`, `tar_started()`, `tar_watch_server()`, `tar_watch_ui()`, `tar_watch()`

Examples

```r
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
  tar_dir({ # tar_dir() runs code from a temporary directory.
  tar_script({
    list(
      tar_target(x, seq_len(100)),
      tar_target(y, Sys.sleep(0.1), pattern = map(x))
    ), ask = FALSE)
  px <- tar_make(callr_function = callr::r_bg, reporter = "silent")
  tar_poll()
})
}
```

---

**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**

```r
tar_process(names = NULL, store = targets::tar_config_get("store"))
```
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 or tidyselect helpers like `all_of()` and `starts_with()`. If NULL, all names are selected.

store  Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

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.

See Also

Other data: `tar_load_raw()`, `tar_load()`, `tar_meta()`, `tar_objects()`, `tar_pid()`, `tar_read_raw()`, `tar_read()`

Examples

```r
if (identical(Sys.getenv("TAR_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)
  })
}
```
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",
  store = targets::tar_config_get("store")
)

Arguments

names  Optional, names of the targets. If supplied, tar_progress() only returns progress information on these targets. You can supply symbols or tidyselect helpers like all_of() and starts_with().
fields Optional, names of progress data columns to read. Set to NULL to read all fields.
store   Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

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", "skipped", "canceled", or "errored".

See Also

Other progress: tar_built(), tar_canceled(), tar_errored(), tar_poll(), tar_progress_branches(), tar_progress_summary(), tar_skipped(), tar_started(), tar_watch_server(), tar_watch_ui(), tar_watch()
Examples

```r
if (identical(Sys.getenv("TAR_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_")) # see also all_of()
})
}
```

tar_progress_branches  Tabulate the progress of dynamic branches.

Description

Read a project’s target progress data for the most recent run of the pipeline and display the tabulated status of dynamic branches. Only the most recent record is shown.

Usage

```r
tar_progress_branches(
  names = NULL,
  fields = NULL,
  store = targets::tar_config_get("store")
)
```

Arguments

- **names**: Optional, names of the targets. If supplied, `tar_progress()` only returns progress information on these targets. You can supply symbols or tidyselect helpers like `starts_with()`.
- **fields**: Optional, names of progress data columns to read. Set to `NULL` to read all fields.
- **store**: Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.
**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.

**See Also**

Other progress: `tar_built()`, `tar_canceled()`, `tar_errored()`, `tar_poll()`, `tar_progress_summary()`, `tar_progress()`, `tar_skipped()`, `tar_started()`, `tar_watch_server()`, `tar_watch_ui()`, `tar_watch()`

**Examples**

```r
if (identical(Sys.getenv("TAR_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()
  })
}
```

Arguments

fields Optional, names of progress data columns to read. Set to NULL to read all fields.

store Character of length 1, path to the targets data store. Defaults to tar_config_get("store").
which in turn defaults to _targets/. When you set this argument, the value of
_of tar_config_get("store") is temporarily changed for the current function
call. See tar_config_get() and tar_config_set() for details about how to
set the data store path persistently for a project.

Value

A data frame with one row and the following optional columns that can be selected with fields.
(time is omitted by default.)

- started: number of targets that started and did not (yet) finish.
- built: number of targets that completed without error or cancellation.
- errored: number of targets that threw an error.
- canceled: number of canceled targets (see tar_cancel()).
- since: how long ago progress last changed (Sys.time() -time).
- time: the time when the progress last changed (modification timestamp of the _targets/meta/progress
  file).

See Also

Other progress: tar_built(), tar_canceled(), tar_errored(), tar_poll(), tar_progress_branches().
tar_progress(), tar_skipped(), tar_started(), tar_watch_server(), tar_watch_ui(), tar_watch()

Examples

if (identical(Sys.getenv("TAR_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), error = "continue")
      ), ask = FALSE)
    try(tar_make())
    tar_progress_summary()
  })
}
**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**

```r
 tar_prune(
    callr_function = callr::r,
    callr_arguments = targets::callr_args_default(callr_function),
    envir = parent.frame(),
    script = targets::tar_config_get("script"),
    store = targets::tar_config_get("store")
)
```

**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.

- **envir**  
  An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc.

  The envir argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative envir, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then envir2 will be used.

- **script**  
  Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to _targets.R. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.
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

```r
tar_read(
  name,
  branches = NULL,
  meta = tar_meta(store = store),
  store = targets::tar_config_get("store")
)
```
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.
- **store**: Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

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.

Limited scope

`tar_read()` and `tar_load()` are only for exploratory analysis and literate programming, and `tar_read_raw()` and `tar_load_raw()` are only for exploratory analysis. `targets` automatically loads the correct dependencies into memory when the pipeline is running, so invoking these functions from inside a target is rarely advisable.

See Also

Other data: `tar_load_raw()`, `tar_load()`, `tar_meta()`, `tar_objects()`, `tar_pid()`, `tar_process()`, `tar_read_raw()`

Examples

```r
if (identical(Sys.getenv("TAR_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(x)
  })
}
```

---

**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(store = store),
  store = targets::tar_config_get("store")
)

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 \texttt{tar_meta()}. \texttt{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 \texttt{tar_meta()} beforehand and supply it to the meta argument, then successive calls to \texttt{tar_read()} may run much faster.
store Character of length 1, path to the \texttt{targets} data store. Defaults to \texttt{tar_config_get("store")}, which in turn defaults to \_targets\/. When you set this argument, the value of \texttt{tar_config_get("store")} is temporarily changed for the current function call. See \texttt{tar_config_get()} and \texttt{tar_config_set()} for details about how to set the data store path persistently for a project.

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.

Limited scope

tar_read() and tar_load() are only for exploratory analysis and literate programming, and tar_read_raw() and tar_load_raw() are only for exploratory analysis. targets automatically loads the correct dependencies into memory when the pipeline is running, so invoking these functions from inside a target is rarely advisable.

See Also

Other data: tar_load_raw(), tar_load(), tar_meta(), tar_objects(), tar_pid(), tar_process(), tar_read()

Examples

if (identical(Sys.getenv("TAR_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 `_targets_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 reenv can identify them automatically.

**Usage**

```r
.tar_renv(
  extras = c("bs4Dash", "clustermq", "future", "gt", "markdown", "pingr", "rstudioapi",
              "shiny", "shinybusy", "shinyWidgets", "visNetwork"),
  path = "/_targets_packages.R",
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script")
)
```

**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`.
- `envir` A list of arguments to `callr_function`.

An environment, where to run the target R script (default: `_targets.R`) if `callr_function` is `NULL`. Ignored if `callr_function` is anything other than `NULL`. `callr_function` should only be `NULL` for debugging and testing purposes, not for serious runs of a pipeline, etc.

The `envir` argument of `tar_make()` and related functions always overrides the current value of `tar_option_get("envir")` in the current R session just before running the target script file, so whenever you need to set an alternative `envir`, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir2, callr_function = NULL)`, then `envir2` will be used.
script Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. When you set this argument, the value of tar_config_get("script") is temporarily changed for the current function call. See tar_script(), tar_config_get(), and tar_config_set() for details about the target script file and how to set it persistently for a project.

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

Other scripts: tar_edit(), tar_github_actions(), tar_helper_raw(), tar_helper(), tar_script()

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("_targets_packages.R"))
})
tar_option_reset()
Description

Create a resources argument for `tar_target()` or `tar_option_set()`.

Usage

tar_resources(
  aws = NULL,
  clustermq = NULL,
  feather = NULL,
  fst = NULL,
  future = NULL,
  parquet = NULL,
  qs = NULL,
  url = NULL
)

Arguments

- **aws**: Output of function `tar_resources_aws()`. AWS S3 storage settings for AWS backed storage formats such as "aws_qs" and "aws_parquet. Applies to all formats beginning with the "aws_" prefix. For details on formats, see the format argument of `tar_target()`.

- **clustermq**: Output of function `tar_resources_clustermq()`. Optional clustermq settings for `tar_make_clustermq()`, including the `log_worker` and `template` arguments of `clustermq::workers()`.

- **feather**: Output of function `tar_resources_feather()`. Non-default arguments to `arrow::read_feather()` and `arrow::write_feather()` for arrow/feather-based storage formats. Applies to all formats ending with the "_feather" suffix. For details on formats, see the format argument of `tar_target()`.

- **fst**: Output of function `tar_resources_fst()`. Non-default arguments to `fst::read_fst()` and `fst::write_fst()` for fst-based storage formats. Applies to all formats ending with "fst" in the name. For details on formats, see the format argument of `tar_target()`.

- **future**: Output of function `tar_resources_future()`. Optional future settings for `tar_make_future()`, including the `resources` argument of `future::future()`, which can include values to insert in template placeholders in `future.batchtools` template files. This is how to supply the `resources` argument of `future::future()` for targets. Resources supplied through `future::plan()` and `future::tweak()` are completely ignored.

- **parquet**: Output of function `tar_resources_parquet()`. Non-default arguments to `arrow::read_parquet()` and `arrow::write_parquet()` for arrow/parquet-based storage formats. Applies to all formats ending with the "_parquet" suffix. For details on formats, see the format argument of `tar_target()`.
Output of function `tar_resources_qs()`.
Non-default arguments to `qs::qread()` and `qs::qsave()` for qs-based storage formats. Applies to all formats ending with the ",_qs" suffix. For details on formats, see the format argument of `tar_target()`.

Output of function `tar_resources_url()`.
Non-default settings for storage formats ending with the ,_url" suffix. These settings include the curl handle for extra control over HTTP requests. For details on formats, see the format argument of `tar_target()`.

**Value**

A list of objects of class "tar_resources" with non-default settings of various optional backends for data storage and high-performance computing.

**Resources**

Functions `tar_target()` and `tar_option_set()` each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner. Resources are all-or-nothing: if you specify any resources with `tar_target()`, all the resources from `tar_option_get("resources")` are dropped for that target. In other words, if you write `tar_option_set(resources = resources_1)` and then `tar_target(x,my_command(),resources = resources_2)`, then everything in `resources_1` is discarded for target `x`.

**See Also**

Other resources: `tar_resources_aws()`, `tar_resources_clusterq()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_url()`

**Examples**

```r
# Somewhere in you target script file (usually _targets.R):

  tar_target(
    name,
    command(),
    format = "qs",
    resources = tar_resources(
      qs = tar_resources_qs(preset = "fast"),
      future = tar_resources_future(resources = list(n_cores = 1))
    )
  )
```
Description

Create the `aws` argument of `tar_resources()` to specify optional settings to AWS storage formats. See the `format` argument of `tar_target()` for details.

Usage

```r
# tar_resources_aws function

tar_resources_aws(bucket, prefix = targets::path_objects_dir_cloud())
```

Arguments

- `bucket`: Character of length 1, name of an existing AWS S3 bucket to upload and download the return values of the affected targets during the pipeline.
- `prefix`: Character of length 1, "directory path" in the S3 bucket where the target return values are stored.

Value

Object of class "tar_resources_aws", to be supplied to the `aws` argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner. Resources are all-or-nothing: if you specify any resources with `tar_target()`, all the resources from `tar_option_get("resources")` are dropped for that target. In other words, if you write `tar_option_set(resources = resources_1)` and then `tar_target(x, my_command(), resources = resources_2)`, then everything in `resources_1` is discarded for target `x`.

See Also

Other resources: `tar_resources_clusterq()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_url()`, `tar_resources()`

Examples

```r
# Somewhere in your target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "aws_qs",
  resources = tar_resources(
    aws = tar_resources_aws(bucket = "yourbucketname"),
  ),
  
```
```r
qs = tar_resources_qs(preset = "fast")
)
)
```

---

**tar_resources_clustermq**

*Target resources: clustermq high-performance computing*

---

**Description**

Create the `clustermq` argument of `tar_resources()` to specify optional high-performance computing settings for `tar_make_clustermq()`. For details, see the documentation of the `clustermq` R package and the corresponding argument names in this help file.

---

**Usage**

```r
tar_resources_clustermq(template = list())
```

---

**Arguments**

- `template` Named list, `template` argument to `clustermq::workers()`.

---

**Value**

Object of class "tar_resources_clustermq", to be supplied to the `clustermq` argument of `tar_resources()`.

---

**Resources**

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner. Resources are all-or-nothing: if you specify any resources with `tar_target()`, all the resources from `tar_option_get("resources")` are dropped for that target. In other words, if you write `tar_option_set(resources = resources_1) and then tar_target(x,my_command(),resources = resources_2)`, then everything in `resources_1` is discarded for target `x`.

---

**See Also**

Other resources: `tar_resources_aws()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_url()`, `tar_resources()`
**Examples**

```r
# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  resources = tar_resources(
    clustermq = tar_resources_clustermq(template = list(n_cores = 2))
  )
)
```

---

**Description**

Create the feather argument of `tar_resources()` to specify optional settings for feather data frame storage formats powered by the `arrow` R package. See the `format` argument of `tar_target()` for details.

**Usage**

```r
tar_resources_feather(compression = "default", compression_level = NULL)
```

**Arguments**

- `compression` Character of length 1, compression argument of `arrow::write_feather()`.
- `compression_level` Numeric of length 1, compression_level argument of `arrow::write_feather()`.

**Value**

Object of class "tar_resources_feather", to be supplied to the feather argument of `tar_resources()`.

**Resources**

Functions `tar_target()` and `tar_option_set()` each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner. Resources are all-or-nothing: if you specify any resources with `tar_target()`, all the resources from `tar_option_get("resources")` are dropped for that target. In other words, if you write `tar_option_set(resources = resources_1)` and then `tar_target(x, my_command(), resources = resources_2)`, then everything in `resources_1` is discarded for target `x`.

**See Also**

Other resources: `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_url()`, `tar_resources()`
Examples

```r
# Somewhere in your target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "feather",
  resources = tar_resources(
    feather = tar_resources_feather(compression = "lz4")
  )
)
```

---

tar_resources_fst  Target resources: fst storage formats

Description

Create the `fst` argument of `tar_resources()` to specify optional settings for big data frame storage formats powered by the `fst` R package. See the `format` argument of `tar_target()` for details.

Usage

```r
tar_resources_fst(compress = 50)
```

Arguments

- `compress` Numeric of length 1, `compress` argument of `fst::write_fst()`.

Value

Object of class "tar_resources_fst", to be supplied to the `fst` argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner. Resources are all-or-nothing: if you specify any resources with `tar_target()`, all the resources from `tar_option_get("resources")` are dropped for that target. In other words, if you write `tar_option_set(resources = resources_1)` and then `tar_target(x, my_command(), resources = resources_2)`, then everything in `resources_1` is discarded for target `x`.

See Also

Other resources: `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_feather()`, `tar_resources_future()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_url()`, `tar_resources()`
Examples

# Somewhere in you target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "fst_tbl",
  resources = tar_resources(
    fst = tar_resources_fst(compress = 100)
  )
)

---

tar_resources_future  Target resources: future high-performance computing

Description

Create the `future` argument of `tar_resources()` to specify optional high-performance computing settings for `tar_make_future()`. This is how to supply the `resources` argument of `future::future()` for targets. Resources supplied through `future::plan()` and `future::tweak()` are completely ignored. For details, see the documentation of the `future` R package and the corresponding argument names in this help file.

Usage

tar_resources_future(plan = NULL, resources = list())

Arguments

- **plan**: A `future::plan()` object or `NULL`, a target-specific future plan.
- **resources**: Named list, `resources` argument to `future::future()`.

Value

Object of class "tar_resources_future", to be supplied to the `future` argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner. Resources are all-or-nothing: if you specify any resources with `tar_target()`, all the resources from `tar_option_get("resources")` are dropped for that target. In other words, if you write `tar_option_set(resources = resources_1)` and then `tar_target(x,my_command(),resources = resources_2)`, then everything in `resources_1` is discarded for target `x`. 
See Also

Other resources: `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_url()`, `tar_resources()`

Examples

```r
# Somewhere in your target script file (usually _targets.R):

# tar_target(
#   name,
#   command(),
#   resources = tar_resources(
#     future = tar_resources_future(resources = list(n_cores = 2))
#   )
# )
```

## tar_resources_parquet

**Target resources: parquet storage formats**

### Description

Create the `parquet` argument of `tar_resources()` to specify optional settings for parquet data frame storage formats powered by the `arrowR` package. See the `format` argument of `tar_target()` for details.

### Usage

```r
tar_resources_parquet(compression = "snappy", compression_level = NULL)
```

### Arguments

- **compression**  
  Character of length 1, `compression` argument of `arrow::write_parquet()`.

- **compression_level**  
  Numeric of length 1, `compression_level` argument of `arrow::write_parquet()`.

### Value

Object of class "tar_resources_parquet", to be supplied to the `parquet` argument of `tar_resources()`.

### Resources

Functions `tar_target()` and `tar_option_set()` each take an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner. Resources are all-or-nothing: if you specify any resources with `tar_target()`, all the resources from `tar_option_get("resources")` are dropped for that target. In other words, if you write `tar_option_set(resources = resources_1)` and then `tar_target(x, my_command(), resources = resources_2)`, then everything in `resources_1` is discarded for target `x`. 
See Also

Other resources: `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_qs()`, `tar_resources_url()`, `tar_resources()`

Examples

```r
# Somewhere in you target script file (usually _targets.R):

tar_target(
  name,
  command(),
  format = "parquet",
  resources = tar_resources(
    parquet = tar_resources_parquet(compression = "lz4")
  )
)
```

---

**tar_resources_qs**

Target resources: qs storage formats

Description

Create the `qs` argument of `tar_resources()` to specify optional settings for big data storage formats powered by the `qs` R package. See the `format` argument of `tar_target()` for details.

Usage

```r
tar_resources_qs(preset = "high")
```

Arguments

- `preset` Character of length 1, preset argument of `qs::qsave()`.

Value

Object of class "tar_resources_qs", to be supplied to the `qs` argument of `tar_resources()`.

Resources

Functions `tar_target()` and `tar_option_set()` each takes an optional resources argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner. Resources are all-or-nothing: if you specify any resources with `tar_target()`, all the resources from `tar_option_get("resources")` are dropped for that target. In other words, if you write `tar_option_set(resources = resources_1)` and then `tar_target(x, my_command(), resources = resources_2)`, then everything in `resources_1` is discarded for target `x`. 
See Also

Other resources: `tar_resources_aws()`, `tar_resources_clustermq()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_parquet()`, `tar_resources_url()`, `tar_resources()`

Examples

```r
# Somewhere in your target script file (usually _targets.R):
tar_target(
  name,
  command(),
  format = "qs",
  resources = tar_resources(
    qs = tar_resources_qs(preset = "fast")
  )
)
```

### tar_resources_url

**Target resources: URL storage formats**

**Description**

Create the `url` argument of `tar_resources()` to specify optional settings for URL storage formats. See the `format` argument of `tar_target()` for details.

**Usage**

`tar_resources_url(handle = NULL)`

**Arguments**

- `handle` Object returned by `curl::new_handle` or `NULL`.

**Value**

Object of class "tar_resources_url", to be supplied to the `url` argument of `tar_resources()`.

**Resources**

Functions `tar_target()` and `tar_option_set()` each takes an optional `resources` argument to supply non-default settings of various optional backends for data storage and high-performance computing. The `tar_resources()` function is a helper to supply those settings in the correct manner. Resources are all-or-nothing: if you specify any resources with `tar_target()`, all the resources from `tar_option_get("resources")` are dropped for that target. In other words, if you write `tar_option_set(resources = resources_1)` and then `tar_target(x,my_command(),resources = resources_2)`, then everything in `resources_1` is discarded for target `x`. 
See Also

Other resources: tar_resources_aws(), tar_resources_clustermq(), tar_resources_feather(), tar_resources_fst(), tar_resources_future(), tar_resources_parquet(), tar_resources_qs(), tar_resources()

Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  # Somewhere in you target script file (usually _targets.R):
  tar_target(
    name,
    command(),
    format = "url",
    resources = tar_resources(
      url = tar_resources_url(handle = curl::new_handle())
    )
  )
}
```

```r
Write a target script file.
```

Description

The `tar_script()` function is a convenient way to create the required target script file (default: _targets.R) 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

```r
tar_script(
  code = NULL,
  library_targets = TRUE,
  ask = NULL,
  script = targets::tar_config_get("script")
)
```

Arguments

- **code**: R code to write to the target script file. If NULL, an example target script file is written instead.
- **library_targets**: logical, whether to write a `library(targets)` line at the top of the target script file automatically (recommended). If TRUE, you do not need to explicitly put `library(targets)` in code.
ask Logical, whether to ask before writing if the target script 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().

script Character of length 1, where to write the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R.

Value

NULL (invisibly).

Target script file

Every targets project requires a target script file. The target script file is usually a file called _targets.R. Functions tar_make() and friends look for the target script and run it to set up the pipeline just prior to the main task. Every target script 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 the target script file 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 target script file 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.

See Also

Other scripts: tar_edit(), tar_github_actions(), tar_helper_raw(), tar_helper(), tar_renv()

Examples

tar_dir({ # tar_dir() runs code from a temporary directory.
tar_script() # Writes an example target script file.
# 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`.

**See Also**
Other utilities: `tar_cancel()`, `tar_envir()`, `tar_group()`, `tar_name()`, `tar_path()`

**Examples**
```
tar_seed()
tar_seed(default = 123L)
if (identical(Sys.getenv("TAR_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)
})
}
```
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

```r
tar_sitrep(
  names = NULL,
  fields = NULL,
  shortcut = targets::tar_config_get("shortcut"),
  reporter = targets::tar_config_get("reporter_outdated"),
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function, reporter),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

Arguments

- **names**
  - Optional, names of the targets. If supplied, `tar_sitrep()` only returns metadata on these targets. You can supply symbols 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 or tidyselect helpers like `all_of()` and `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.

**shortcut**  
Logical of length 1, how to interpret the names argument. If shortcut is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. If TRUE, then the function only checks the targets in names and uses stored metadata for information about upstream dependencies as needed. shortcut = TRUE increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Use with caution. shortcut = TRUE only works if you set names.

**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.

**envir**  
An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc.

The envir argument of tar_make() and related functions always overrides the current value of tar_option_get("envir") in the current R session just before running the target script file, so whenever you need to set an alternative envir, you should always set it with tar_option_set() from within the target script file. In other words, if you call tar_option_set(envir = envir1) in an interactive session and then tar_make(envir = envir2, callr_function = NULL), then envir2 will be used.

**script**  
Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. When you set this argument, the value of tar_config_get("script") is temporarily changed for the current function
call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

**store**

Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**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.

**See Also**

Other inspect: `tar_deps_raw()`, `tar_deps()`, `tar_glimpse()`, `tar_manifest()`, `tar_network()`, `tar_outdated()`, `tar_validate()`, `tar_visnetwork()`

**Examples**

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({
    tar_script({
      list(
        tar_target(x, seq_len(2)),
        tar_target(y, 2 * x, pattern = map(x))
      ),
      ask = FALSE
    )
  }, ask = FALSE)
  tar_make()
  tar_sitrep()
  tar_meta(starts_with("y_"))  # see also all_of()
}
```
**tar_skipped**

List skipped targets.

**Description**

List targets whose progress is "skipped".

**Usage**

```r
tar_skipped(names = NULL, store = targets::tar_config_get("store"))
```

**Arguments**

- `names` Optional, names of the targets. If supplied, the function restricts its output to these targets. You can supply symbols or tidyselect helpers like `all_of()` and `starts_with()`.
- `store` Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `.targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Value**

A character vector of skipped targets.

**See Also**

Other progress: `tar_built()`, `tar_canceled()`, `tar_errored()`, `tar_poll()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_progress()`, `tar_started()`, `tar_watch_server()`, `tar_watch_ui()`, `tar_watch()`

**Examples**

```r
if (identical(Sys.getenv("TAR_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_skipped()
    tar_skipped(starts_with("y_")) # see also all_of()
  })
}
```
**Description**

List targets whose progress is "started".

**Usage**

```
tar_started(names = NULL, store = targets::tar_config_get("store"))
```

**Arguments**

- `names` Optional, names of the targets. If supplied, the function restricts its output to these targets. You can supply symbols or tidyselect helpers like `all_of()` and `starts_with()`.
- `store` Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Value**

A character vector of started targets.

**See Also**

Other progress: `tar_built()`, `tar_canceled()`, `tar_errored()`, `tar_poll()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_progress()`, `tar_skipped()`, `tar_watch_server()`, `tar_watch_ui()`, `tar_watch()`

**Examples**

```r
if (identical(Sys.getenv("TAR_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_started()
    tar_started(starts_with("y")) # see also all_of()
  })
}
```
**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. See the "Storage formats" section for a detailed list of possible data storage formats.

### 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 stops and throws an error. Options:
- "stop": the whole pipeline stops and throws an error.
- "continue": the whole pipeline keeps going.
- "abridge": any currently running targets keep running, but no new targets launch after that. (Visit [https://books.ropensci.org/targets/debugging.html](https://books.ropensci.org/targets/debugging.html) to learn how to debug targets using saved workspaces.)

### 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 strategy 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()`).

resources: Object returned by `tar_resources()` with optional settings for high-performance computing functionality, alternative data storage formats, and other optional capabilities of targets. See `tar_resources()` for details.

storage: Character of length 1, only relevant to `tar_make_cluster()` 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_cluster()` 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 target script file (default: `_targets.R`).

Storage 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 `tar_resources()` and `tar_resources_qs()`.
- "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 `tar_resources()` and `tar_resources_feather()`. 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 `tar_resources()` and `tar_resources_parquet()`. 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 `tar_resources()` and `tar_resources_fst()`. 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. Option-
ally supply a custom curl handle through tar_resources() and tar_resources_url(). in
new_handle(), nobody = TRUE is important because it ensures targets just downloads the
metadata instead of the entire data file when it checks time stamps and hashes. 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 tar_resources_aws(). 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 tem-
porary 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.

See Also

Other targets: tar_cue(), tar_target_raw()

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_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(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  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. See the "Storage formats" section for a detailed list of possible data storage formats.

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 stops and throws an error. Options:

- "stop": the whole pipeline stops and throws an error.
• "continue": the whole pipeline keeps going.
• "abridge": any currently running targets keep running, but no new targets launch after that. (Visit https://books.ropensci.org/targets/debugging.html to learn how to debug targets using saved workspaces.)

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 strategy 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()).

resources
Object returned by tar_resources() with optional settings for high-performance computing functionality, alternative data storage formats, and other optional capabilities of targets. See tar_resources() 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 target script file (default: _targets.R).

See Also
Other targets: tar_cue(), tar_target()
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_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

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

Arguments

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

Value

`NULL` (invisibly).

See Also

Other utilities to extend targets: `tar_assert`, `tar_condition`, `tar_dir()`, `tar_language`
Examples

```r
  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_timestamp**

*Get the timestamp(s) of a target.*

**Description**

Get the time that a target last ran successfully. If there are multiple artifacts, as with file or URL targets, then multiple time stamps may be returned.

**Usage**

```r
tar_timestamp(
  name = NULL,
  format = NULL,
  tz = NULL,
  parse = NULL,
  store = targets::tar_config_get("store")
)
```

**Arguments**

- **name**
  
  Symbol, name of the target. If NULL (default) then `tar_timestamp()` will attempt to return the timestamp of the target currently running. Must be called inside a target’s command or a supporting function in order to work.

- **format**
  
  Deprecated in `targets` version 0.6.0 (2021-07-21).

- **tz**
  
  Deprecated in `targets` version 0.6.0 (2021-07-21).

- **parse**
  
  Deprecated in `targets` version 0.6.0 (2021-07-21).

- **store**
  
  Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Details**

`tar_timestamp()` checks the metadata in `_targets/meta/meta`, not the actual data. Time stamps are recorded only for targets that run commands: just non-branching targets and individual dynamic branches.
Value

If the target is not recorded in the metadata or cannot be parsed correctly, then \texttt{tar_timestamp()} returns a \texttt{POSIXct} object at 1970-01-01 UTC.

See Also

Other time: \texttt{tar_newer()}, \texttt{tar_older()}, \texttt{tar_timestamp_raw()}

Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir(# tar_dir() runs code from a temporary directory.
  tar_script({
    list(tar_target(x, 1))
  }, ask = FALSE)
  tar_make()
  # Get the timestamp.
  tar_timestamp(x)
  # We can use the timestamp to cancel the target
  # if it already ran within the last hour.
  # Be sure to set `cue = tar_cue(mode = "always")`
  # if you want the target to always check the timestamp.
  tar_script({
    list(
      tar_target(
        x,
        tar_cancel((Sys.time() - tar_timestamp()) < 3600),
        cue = tar_cue(mode = "always")
      )
    ), ask = FALSE)
  tar_make()
  })
}
```

---

\texttt{tar_timestamp_raw} \hspace{1cm} \textit{Get the timestamp(s) of a target (raw version).}

Description

Get the time that a target last ran successfully.

Usage

```r
\texttt{tar_timestamp_raw(}
  name = NULL,
  format = NULL,
  tz = NULL,
  parse = NULL,
  store = targets::tar_config_get("store")
\)}
```
Arguments

name  Character of length 1, name of the target.
format  Deprecated in targets version 0.6.0 (2021-07-21).
tz  Deprecated in targets version 0.6.0 (2021-07-21).
parse  Deprecated in targets version 0.6.0 (2021-07-21).
store  Character of length 1, path to the targets data store. Defaults to \texttt{tar_config_get("store")}, which in turn defaults to \_targets\. When you set this argument, the value of \texttt{tar_config_get("store")} is temporarily changed for the current function call. See \texttt{tar_config_get()} and \texttt{tar_config_set()} for details about how to set the data store path persistently for a project.

Details

\texttt{tar_timestamp_raw()} is like \texttt{tar_timestamp()} except it accepts a character string for \texttt{name} instead of a symbol. \texttt{tar_timestamp_raw()} checks the metadata in \_targets/meta/meta, not the actual data. Time stamps are recorded only for targets that run commands: just non-branching targets and individual dynamic branches.

Value

If the target is not recorded in the metadata or cannot be parsed correctly, then \texttt{tar_timestamp_raw()} returns a \texttt{POSIXct} object at 1970-01-01 UTC.

See Also

Other time: \texttt{tar_newer()}, \texttt{tar_older()}, \texttt{tar_timestamp()}

Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({ # \texttt{tar_dir()} runs code from a temporary directory.
    tar_script({
      list(tar_target(x, 1))
    }, ask = FALSE)
    tar_make()
    # Get the timestamp.
    tar_timestamp_raw("x")
    # We can use the timestamp to cancel the target
    # if it already ran within the last hour.
    # Be sure to set "cue = tar_cue(mode = "always")"
    # if you want the target to always check the timestamp.
    tar_script({
      list(
        tar_target(
          x,
          tar_cancel((Sys.time() - tar_timestamp_raw()) < 3600),
          cue = tar_cue(mode = "always")
        )
      ), ask = FALSE)
```
**tar_toggle**

Choose code to run based on Target Markdown mode.

**Description**
Run one piece of code if Target Markdown mode interactive mode is turned on and another piece of code otherwise.

**Usage**

```r
.tar_toggle(interactive, noninteractive)
```

**Arguments**

- `interactive` R code to run if Target Markdown interactive mode is activated.
- `noninteractive` R code to run if Target Markdown interactive mode is not activated.

**Details**
Visit <books.ropensci.org/targets/markdown.html> to learn about Target Markdown and interactive mode.

**Value**
If Target Markdown interactive mode is not turned on, the function returns the result of running the code. Otherwise, the function invisibly returns `NULL`.

**See Also**
Other Target Markdown: `tar_engine_knitr()`, `tar_interactive()`, `tar_noninteractive()`

**Examples**

```r
tax_toggle(
  message("In interactive mode."),
  message("Not in interactive mode.")
)
```
Description

Return the saved traceback of a target. Assumes the target errored out in a previous run of the pipeline with workspaces enabled for that target. See `tar_workspace()` for details.

Usage

```r
tar_traceback(
  name,
  envir = NULL,
  packages = NULL,
  source = NULL,
  characters =getOption("width"),
  store = targets::tar_config_get("store")
)
```

Arguments

- **name**: Symbol, name of the target whose workspace to read.
- **envir**: Deprecated in `targets > 0.3.1` (2021-03-28).
- **packages**: Logical, whether to load the required packages of the target.
- **source**: Logical, whether to run the target script file (default: `_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.
- **characters**: Positive integer. Each line of the traceback is shortened to this number of characters.
- **store**: Character of length 1, path to the `targets` data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

Value

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

See Also

Other debug: `tar_load Globals()`, `tar_workspaces()`, `tar_workspace()`
Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({ # tar_dir() runs code from a temporary directory.
    tmp <- sample(1)
  }
  tar_script({
    tar_option_set(workspace_on_error = TRUE)
    list(
      tar_target(x, "loaded"),
      tar_target(y, stop(x))
    ),
    ask = FALSE)
  try(tar_make())
  tar_traceback(y, characters = 60)
})
}
```

---

**tar_unscript**

Remove target script helper files.

---

**Description**

Target Markdown writes helper scripts in a folder accompanying the target script file. If the target script is the default _targets.R, the helper scripts are in _targets_r/. If the some other location like custom/script.R, the helper scripts are in custom/script_r/. Use `tar_unscript()` to remove all these helper scripts in the *_.r/ directory. The actual target script is not removed.

**Usage**

```r
tar_unscript(script = targets::tar_config_get("script"))
```

**Arguments**

- **script**
  
  Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to _targets.R. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

**Value**

NULL (invisibly).

**Examples**

```r
  tar_dir({ # tar_dir() runs code from a temporary directory.
    tar_unscript()
  })
```
**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**

```r
.tar_validate(
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```

**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**.

- **envir**: An environment, where to run the target R script (default: _targets.R) if **callr_function** is NULL. Ignored if **callr_function** is anything other than NULL. **callr_function** should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc.

  The `envir` argument of **tar_make()** and related functions always overrides the current value of **tar_option_get("envir")** in the current R session just before running the target script file, so whenever you need to set an alternative envir, you should always set it with `tar_option_set()` from within the target script file. In other words, if you call `tar_option_set(envir = envir1)` in an interactive session and then `tar_make(envir = envir2, callr_function = NULL)`, then `envir2` will be used.

- **script**: Character of length 1, path to the target script file. Defaults to `tar_config_get("script")`, which in turn defaults to _targets.R. When you set this argument, the value of `tar_config_get("script")` is temporarily changed for the current function call. See `tar_script()`, `tar_config_get()`, and `tar_config_set()` for details about the target script file and how to set it persistently for a project.

- **store**: Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to _targets/. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call.
call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**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.

**See Also**

Other `inspect`: `tar_deps_raw()`, `tar_deps()`, `tar_glimpse()`, `tar_manifest()`, `tar_network()`, `tar_outdated()`, `tar_sitrep()`, `tar_visnetwork()`

**Examples**

```r
if (identical(Sys.getenv("TAR_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 the target script file (default: `targets.R`) and visualize the directed acyclic graph of targets and global functions and objects.

**Usage**

```r
tar_visnetwork(
  targets_only = FALSE,
  names = NULL,
  shortcut = FALSE,
  allow = NULL,
  exclude = ".Random.seed",
  outdated = TRUE,
  label = NULL,
  level_separation = NULL,
  degree_from = 1L,
  degree_to = 1L,
  reporter = targets::tar_config_get("reporter_outdated"),
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function),
  envir = parent.frame(),
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```
Arguments

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

**names** Names of targets. The graph visualization will operate only on these targets (and unless shortcut is TRUE, all the targets upstream as well). Selecting a small subgraph using names could speed up the load time of the visualization. Unlike allow, names is invoked before the graph is generated. Set to NULL to check/build all the targets (default). Otherwise, you can supply symbols or tidyselect helpers like starts_with(). Applies to ordinary targets (stem) and whole dynamic branching targets (patterns) but not individual dynamic branches.

**shortcut** Logical of length 1, how to interpret the names argument. If shortcut is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. If TRUE, then the function only checks the targets in names and uses stored metadata for information about upstream dependencies as needed. shortcut = TRUE increases speed if there are a lot of up-to-date targets, but it assumes all the dependencies are up to date, so please use with caution. Also, shortcut = TRUE only works if you set names.

**allow** Optional, define the set of allowable vertices in the graph. Unlike names, allow is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to NULL to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols or tidyselect helpers like starts_with().

**exclude** Optional, define the set of exclude vertices from the graph. Unlike names, exclude is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to NULL to exclude no vertices. Otherwise, you can supply symbols or tidyselect helpers like all_of() and 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.

**degree_from** Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. degree_from controls the number of edges the neighborhood extends upstream.

**degree_to** Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. degree_to controls the number of edges the neighborhood extends downstream.
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.

evr An environment, where to run the target R script (default: _targets.R) if callr_function is NULL. Ignored if callr_function is anything other than NULL. callr_function should only be NULL for debugging and testing purposes, not for serious runs of a pipeline, etc. The envir argument of tar_make() and related functions always overrides the current value of tar_option_get("envir") in the current R session just before running the target script file, so whenever you need to set an alternative envir, you should always set it with tar_option_set() from within the target script file. In other words, if you call tar_option_set(envir = envir1) in an interactive session and then tar_make(envir = envir2, callr_function = NULL), then envir2 will be used.

script Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. When you set this argument, the value of tar_config_get("script") is temporarily changed for the current function call. See tar_script(), tar_config_get(), and tar_config_set() for details about the target script file and how to set it persistently for a project.

store Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_script() and tar_config_set() for details about how to set the data store path persistently for a project.

Value
A visNetwork HTML widget object.

See Also
Other inspect: tar_deps_raw(), tar_deps(), tar_glimpse(), tar_manifest(), tar_network(), tar_outdated(), tar_sitrep(), tar_validate()

Examples
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {

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 = 10,
  seconds_min = 1,
  seconds_max = 60,
  seconds_step = 1,
  targets_only = FALSE,
  exclude = ".Random.seed",
  outdated = FALSE,
  label = NULL,
  level_separation = 150,
  degree_from = 1L,
  degree_to = 1L,
  config = ".targets.yaml",
  height = "650px",
  display = "summary",
  displays = c("summary", "branches", "progress", "graph", "about"),
  background = TRUE,
  browse = TRUE,
  host =getOption("shiny.host", "127.0.0.1"),
  port =getOption("shiny.port", targets::tar_random_port()),
  verbose = TRUE,
  supervise = TRUE,
  poll_connection = TRUE,
  stdout = "|",
)
stderr = "|"
)

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 global functions and objects.
exclude Character vector of nodes to omit from the graph.
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.
degree_from Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. degree_from controls the number of edges the neighborhood extends upstream.
degree_to Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. degree_to controls the number of edges the neighborhood extends downstream.
config Character of length 1, path to the YAML file with all the configuration settings (default: _targets.yaml).
height Character of length 1, height of the visNetwork widget and branches table.
display Character of length 1, which display to show first.
displays Character vector of choices for the display. Elements can be any of "graph", "summary", "branches", or "about".
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.

### poll_connection
Whether to have a control connection to the process. This is used to transmit messages from the subprocess to the main process.

### stdout
The name of the file the standard output of the child R process will be written to. If the child process runs with the \--slave option (the default), then the commands are not echoed and will not be shown in the standard output. Also note that you need to call print() explicitly to show the output of the command(s).

### stderr
The name of the file the standard error of the child R process will be written to. In particular message() sends output to the standard error. If nothing was sent to the standard error, then this file will be empty. This argument can be the same file as stdout, in which case they will be correctly interleaved. If this is the string "2>&1", then standard error is redirected to standard output.

### 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.

### See Also
Other progress: `tar_built()`, `tar_canceled()`, `tar_erred()`, `tar_poll()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_progress()`, `tar_skipped()`, `tar_started()`, `tar_watch_server()`, `tar_watch_ui()`

### Examples
```r
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()
}
Description

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

Usage

```r
tar_watch_server(
  id,
  height = "650px",
  exclude = ".Random.seed",
  config = "_targets.yaml"
)
```

Arguments

- `id` Character of length 1, ID corresponding to the UI function of the module.
- `height` Character of length 1, height of the `visNetwork` widget and branches table.
- `exclude` Character vector of nodes to omit from the graph.
- `config` Character of length 1, path to the YAML file with all the configuration settings (default: `_targets.yaml`).

Value

A Shiny module server.

See Also

Other progress: `tar_built()`, `tar_canceled()`, `tar_errored()`, `tar_poll()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_progress()`, `tar_skipped()`, `tar_started()`, `tar_watch_ui()`, `tar_watch()`
Description

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

Usage

```r
.tar_watch_ui(
  id,
  label = "tar_watch_label",
  seconds = 10,
  seconds_min = 1,
  seconds_max = 60,
  seconds_step = 1,
  targets_only = FALSE,
  outdated = FALSE,
  label_tar_visnetwork = NULL,
  level_separation = 150,
  degree_from = 1L,
  degree_to = 1L,
  height = "650px",
  display = "summary",
  displays = c("summary", "branches", "progress", "graph", "about")
)
```

Arguments

- **id** Character of length 1, ID corresponding to the UI function of the module.
- **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 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
Characer 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.

degree_from
Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_from` controls the number of edges the neighborhood extends upstream.

degree_to
Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. `degree_to` controls the number of edges the neighborhood extends downstream.

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

display
Character of length 1, which display to show first.

displays
Character vector of choices for the display. Elements can be any of "graph", "summary", "branches", or "about".

Value
A Shiny module UI.

See Also
Other progress: `tar_built()`, `tar_canceled()`, `tar_errored()`, `tar_poll()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_progress()`, `tar_skipped()`, `tar_started()`, `tar_watch_server()`, `tar_watch()`

tar_workspace

Description
Load a saved workspace and seed for debugging.

Usage

```r
.tar_workspace(
  name,
  envir = parent.frame(),
  packages = TRUE,
  source = TRUE,
  script = targets::tar_config_get("script"),
  store = targets::tar_config_get("store")
)
```
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.
- **script**: Character of length 1, path to the target script file. Defaults to tar_config_get("script"), which in turn defaults to _targets.R. When you set this argument, the value of tar_config_get("script") is temporarily changed for the current function call. See tar_script(), tar_config_get(), and tar_config_set() for details about the target script file and how to set it persistently for a project.
- **store**: Character of length 1, path to the targets data store. Defaults to tar_config_get("store"), which in turn defaults to _targets/. When you set this argument, the value of tar_config_get("store") is temporarily changed for the current function call. See tar_config_get() and tar_config_set() for details about how to set the data store path persistently for a project.

Details

If you activate workspaces through the workspaces argument of tar_option_set(), then under the circumstances you specify, targets will save a special workspace file to a location in _targets/workspaces/. The workspace file 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().

See Also

Other debug: tar_load Globals(), tar_traceback(), tar_workspaces()

Examples

```r
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({ # tar_dir() runs code from a temporary directory.
    tmp <- sample(1)
    tar_script({
      tar_option_set(workspace_on_error = TRUE)
      list(
        tar_target(x, "loaded"),
```
```r

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, store = targets::tar_config_get("store"))
```

**Arguments**

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

- `store`  
  Character of length 1, path to the targets data store. Defaults to `tar_config_get("store")`, which in turn defaults to `_targets/`. When you set this argument, the value of `tar_config_get("store")` is temporarily changed for the current function call. See `tar_config_get()` and `tar_config_set()` for details about how to set the data store path persistently for a project.

**Value**

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

**See Also**

Other debug: `tar_load_globals()`, `tar_traceback()`, `tar_workspace()`
Examples

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

**use_targets**  
*Use targets*

**Description**

Create an example Target Markdown report to get started with targets.

**Usage**

```r
use_targets(path = "_targets.Rmd", open = interactive())
```

**Arguments**

- `path` Character of length 1, output path of the Target Markdown report relative to the current active project.
- `open` Logical, whether to open the file for editing in the RStudio IDE.

**Value**

`NULL` (invisibly).

**See Also**

Other help: `targets-package`

**Examples**

```r
if (identical(Sys.getenv("TAR_INTERACTIVE_EXAMPLES"), "true")) {
  tar_dir({ # tar_dir() runs code from a temporary directory.
    use_targets(open = FALSE)
  })
}
```
# Index

* **Target Markdown**
  - `tar_engine_knitr`, 24
  - `tar_interactive`, 40
  - `tar_noninteractive`, 63
  - `tar_toggle`, 120

* **branching**
  - `tar_branch_index`, 8
  - `tar_branch_names`, 9
  - `tar_branch_names_raw`, 10
  - `tar_branches`, 7
  - `tar_pattern`, 75

* **clean**
  - `tar_delete`, 19
  - `tar_destroy`, 22
  - `tar_invalidate`, 41
  - `tar_prune`, 85

* **configuration**
  - `tar_config_get`, 14
  - `tar_config_set`, 15
  - `tar_envvars`, 27
  - `tar_option_get`, 66
  - `tar_option_reset`, 67
  - `tar_option_set`, 68

* **data**
  - `tar_load`, 43
  - `tar_load_raw`, 45
  - `tar_meta`, 56
  - `tar_objects`, 64
  - `tar_pid`, 77
  - `tar_process`, 79
  - `tar_read`, 86
  - `tar_read_raw`, 87

* **debug**
  - `tar_load_globals`, 44
  - `tar_traceback`, 121
  - `tar_workspace`, 132
  - `tar_workspaces`, 134

* **existence**
  - `tar_exist_meta`, 29
  - `tar_exist_objects`, 30
  - `tar_exist_process`, 31
  - `tar_exist_progress`, 31
  - `tar_exist_script`, 32

* **help**
  - `targets-package`, 4
  - `use_targets`, 135

* **inspect**
  - `tar_deps`, 20
  - `tar_deps_raw`, 21
  - `tar_glimpse`, 34
  - `tar_manifest`, 54
  - `tar_network`, 59
  - `tar_outdated`, 72
  - `tar_sitrep`, 104
  - `tar_validate`, 123
  - `tar_visnetwork`, 124

* **pipeline**
  - `tar_make`, 47
  - `tar_make_clustermq`, 49
  - `tar_make_future`, 51

* **progress**
  - `tar_built`, 11
  - `tar_canceled`, 13
  - `tar_errored`, 28
  - `tar_poll`, 78
  - `tar_progress`, 81
  - `tar_progress_branches`, 82
  - `tar_progress_summary`, 83
  - `tar_skipped`, 107
  - `tar_started`, 108
  - `tar_watch`, 127
  - `tar_watch_server`, 130
  - `tar_watch_ui`, 131

* **resources**
  - `tar_resources`, 91
  - `tar_resources_aws`, 93
  - `tar_resources_clustermq`, 94
  - `tar_resources_feather`, 95
INDEX

* utilities
  * scripts
    * tar_edit, 24
    * tar_github_actions, 33
    * tar_helper, 38
    * tar_helper_raw, 39
    * tar_renv, 89
    * tar_script, 101
  * targets
    * tar_cue, 17
    * tar_target, 109
    * tar_target_raw, 113
  * time
    * tar_newer, 62
    * tar_older, 65
    * tar_timestamp, 117
    * tar_timestamp_raw, 118
  * utilities to extend targets
    * tar_assert, 5
    * tar_condition, 14
    * tar_dir, 23
    * tar_language, 42
    * tar_test, 116
  * utilities
    * tar_cancel, 12
    * tar_envir, 26
    * tar_group, 36
    * tar_name, 58
    * tar_path, 74
    * tar_seed, 103

all_of(), 11, 13, 19, 28, 35, 41, 43, 47, 49, 52, 54, 56, 60, 62, 64, 65, 72, 80, 81, 104, 107, 108, 125

starts_with(), 11, 13, 19, 28, 35, 41, 43, 47, 49, 52, 54, 56, 60, 62, 64, 65, 72, 80–82, 104, 107, 108, 125

tail (tar_pattern), 75

tar_assert, 5, 14, 23, 42, 116

tar_assert_chr (tar_assert), 5

tar_assert_dbl (tar_assert), 5

tar_assert_df (tar_assert), 5

tar_assert_envir (tar_assert), 5

tar_assert_equal_lengths (tar_assert), 5

tar_assert_expr (tar_assert), 5

tar_assert_flag (tar_assert), 5

tar_assert_function (tar_assert), 5

tar_assert_ge (tar_assert), 5

tar_assert_identical (tar_assert), 5

tar_assert_in (tar_assert), 5

tar_assert_inherits (tar_assert), 5

tar_assert_int (tar_assert), 5

tar_assert_internet (tar_assert), 5

tar_assert_lang (tar_assert), 5

tar_assert_le (tar_assert), 5

tar_assert_lgl (tar_assert), 5

tar_assert_match (tar_assert), 5

tar_assert_names (tar_assert), 5

tar_assert_nonempty (tar_assert), 5

tar_assert_nonmissing (tar_assert), 5

tar_assert_not_dir (tar_assert), 5

tar_assert_not_dirs (tar_assert), 5

tar_assert_not_expr (tar_assert), 5

tar_assert_not_in (tar_assert), 5

tar_assert_nzchar (tar_assert), 5

tar_assert_package (tar_assert), 5

tar_assert_path (tar_assert), 5

tar_assert_positive (tar_assert), 5

tar_assert_scalar (tar_assert), 5

tar_assert_target (tar_assert), 5

tar_assert_target_list (tar_assert), 5

tar_assert_true (tar_assert), 5

tar_assert_unique (tar_assert), 5

tar_assert_unique_targets (tar_assert), 5

tar_branch_index, 8, 8, 10, 76

tar_branch_names, 8, 9, 9, 10, 76

tar_branch_names(), 10

tar_branch_names_raw, 8–10, 10, 76

tar_branches, 7, 9, 10, 76

head(tar_pattern), 75

cross (tar_pattern), 75

library(), 89

list(), 102, 111, 115

map (tar_pattern), 75

sample (tar_pattern), 75
INDEX

| tar_progress, 11, 13, 28, 79, 81, 83, 84, 107, 108, 129, 130, 132 |
| tar_progress(), 58 |
| tar_progress_branches, 11, 13, 28, 79, 81, 82, 84, 107, 108, 129, 130, 132 |
| tar_progress_summary(), 78 |
| tar_prune, 20, 23, 41, 85 |
| tar_read, 44, 46, 58, 64, 78, 80, 86, 88 |
| tar_read(), 87, 112 |
| tar_read_raw, 44, 46, 58, 64, 78, 80, 87, 87 |
| tar_renv, 24, 34, 39, 89, 102 |
| tar_resources, 91, 93–96, 98–101 |
| tar_resources_aws, 92, 93, 94–96, 98–101 |
| tar_resources_cluster, 92, 93, 94, 95, 96, 98–101 |
| tar_resources_father, 92–94, 95, 96, 98–101 |
| tar_resources_fst, 92–95, 96, 98–101 |
| tar_resources_future, 92–96, 97, 99–101 |
| tar_resources_parquet, 92–96, 98, 98, 100, 101 |
| tar_resources_u, 92–96, 98, 99, 101 |
| tar_resources_url, 92–96, 98–100, 100 |
| tar_script, 24, 34, 39, 90, 101 |
| tar_script(), 24, 27, 32, 36, 38, 39, 48, 50, 53, 55, 61, 73, 74, 85, 90, 106, 122, 123, 126, 133 |
| tar_seed, 12, 26, 37, 59, 75, 103 |
| tar_sitrep, 20, 21, 36, 55, 61, 74, 104, 124, 126 |
| tar_skipped, 11, 13, 28, 79, 81, 83, 84, 107, 108, 129, 130, 132 |
| tar_started, 11, 13, 28, 79, 81, 83, 84, 107, 108, 129, 130, 132 |
| tar_target, 19, 109, 115 |
| tar_target(), 18, 54, 57, 66–68, 75, 76, 90–100, 102, 113, 114 |
| tar_target_raw, 19, 112, 113 |
| tar_target_raw(), 54, 66–68 |
| tar_test, 7, 14, 23, 42, 116 |
| tar_throw_file (tar_condition), 14 |
| tar_throw_run (tar_condition), 14 |
| tar_throw_validate (tar_condition), 14 |
| tar_tidy_eval (tar_language), 42 |
| tar_tidyselect_eval (tar_language), 42 |
| tar_timestamp, 62, 65, 117, 119 |
| tar_timestamp_raw, 62, 65, 118, 118 |
| tar_toggle, 26, 40, 63, 120 |
| tar_traceback, 45, 121, 133, 134 |
| tar_unscript, 122 |
| tar_unscript(), 122 |
| tar_validate, 20, 21, 36, 55, 61, 74, 106, 123, 126 |
| tar_visnetwork, 20, 21, 36, 55, 61, 74, 106, 124, 124 |
| tar_visnetwork(), 34, 54, 127–129, 132 |
| tar_warn_deprecate (tar_condition), 14 |
| tar_warn_run (tar_condition), 14 |
| tar_warn_validate (tar_condition), 14 |
| tar_watch(), 130, 131 |
| tar_watch_server(), 127, 131 |
| tar_watch_ui, 11, 13, 28, 79, 81, 83, 84, 107, 108, 129, 130, 131 |
| tar_watch_ui(), 127, 130 |
| tar_workspace, 45, 121, 132, 134 |
| tar_workspace(), 22, 71, 121, 134 |
| tar_workspaces, 45, 121, 133, 134 |
| targets-package, 4 |
| use targets, 4, 135 |