

# Package ‘targets’

September 21, 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](https://doi.org/10.21105/joss.00550)>).

**Version** 0.8.0

**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), knitr (>= 1.34), 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), markdown (>= 1.1), rmarkdown (>= 2.4), pingr (>= 2.0.1), pkgload (>= 1.1.0), qs (>= 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** 2021-09-21 13:30:02 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 . . . . .	16
tar_config_unset . . . . .	19
tar_cue . . . . .	20
tar_definition . . . . .	22
tar_delete . . . . .	24
tar_deps . . . . .	25
tar_deps_raw . . . . .	26
tar_destroy . . . . .	26
tar_dir . . . . .	28
tar_edit . . . . .	28
tar_engine_knitr . . . . .	29
tar_envir . . . . .	31
tar_envvars . . . . .	32
tar_errored . . . . .	33
tar_exist_meta . . . . .	34
tar_exist_objects . . . . .	35
tar_exist_process . . . . .	36
tar_exist_progress . . . . .	36
tar_exist_script . . . . .	37

tar_github_actions . . . . .	38
tar_glimpse . . . . .	39
tar_group . . . . .	41
tar_helper . . . . .	43
tar_helper_raw . . . . .	44
tar_interactive . . . . .	45
tar_invalidate . . . . .	46
tar_language . . . . .	47
tar_load . . . . .	48
tar_load_globals . . . . .	49
tar_load_raw . . . . .	50
tar_make . . . . .	52
tar_make_clustermq . . . . .	54
tar_make_future . . . . .	56
tar_manifest . . . . .	59
tar_meta . . . . .	61
tar_name . . . . .	63
tar_network . . . . .	64
tar_newer . . . . .	67
tar_noninteractive . . . . .	68
tar_objects . . . . .	69
tar_older . . . . .	70
tar_option_get . . . . .	71
tar_option_reset . . . . .	72
tar_option_set . . . . .	73
tar_outdated . . . . .	77
tar_path . . . . .	80
tar_pattern . . . . .	81
tar_pid . . . . .	83
tar_poll . . . . .	84
tar_process . . . . .	85
tar_progress . . . . .	86
tar_progress_branches . . . . .	87
tar_progress_summary . . . . .	89
tar_prune . . . . .	90
tar_read . . . . .	92
tar_read_raw . . . . .	93
tar_renv . . . . .	94
tar_resources . . . . .	96
tar_resources_aws . . . . .	98
tar_resources_clustermq . . . . .	99
tar_resources_feather . . . . .	100
tar_resources_fst . . . . .	101
tar_resources_future . . . . .	102
tar_resources_parquet . . . . .	103
tar_resources_qs . . . . .	104
tar_resources_url . . . . .	105
tar_script . . . . .	106

tar_seed . . . . .	108
tar_sitrep . . . . .	109
tar_skipped . . . . .	112
tar_started . . . . .	113
tar_target . . . . .	114
tar_target_raw . . . . .	119
tar_test . . . . .	122
tar_timestamp . . . . .	123
tar_timestamp_raw . . . . .	125
tar_toggle . . . . .	126
tar_traceback . . . . .	127
tar_unscript . . . . .	128
tar_validate . . . . .	129
tar_visnetwork . . . . .	130
tar_watch . . . . .	133
tar_watch_server . . . . .	137
tar_watch_ui . . . . .	138
tar_workspace . . . . .	139
tar_workspaces . . . . .	141
use_targets . . . . .	142

**Index****144**


---

targets-package	<i>targets: Dynamic Function-Oriented Make-Like Declarative Pipelines for R</i>
-----------------	---

---

**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](https://doi.org/10.21105/joss.00550)).

**See Also**

Other help: [use\\_targets\(\)](#)

---

tar_assert	<i>Assertions</i>
------------	-------------------

---

**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.

path	Character, file path.
pattern	Character of length 1, a grep pattern for certain assertions.

**See Also**

Other utilities to extend targets: [tar\\_condition](#), [tar\\_dir\(\)](#), [tar\\_language](#), [tar\\_test\(\)](#)

**Examples**

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

---

tar_branches	<i>Reconstruct the branch names and the names of their dependencies.</i>
--------------	--

---

**Description**

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

**Usage**

```
tar_branches(name, pattern, 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 <code>x</code> and <code>y</code> , <code>tar_target(z, x + y, pattern = map(x, y))</code> implicitly defines branches of <code>z</code> that each compute <code>x[1] + y[1]</code> , <code>x[2] + y[2]</code> , and so on. See the user manual for details.
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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**

```
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	<i>Integer branch indexes</i>
------------------	-------------------------------

---

**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**

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

**Value**

A named integer vector of branch indexes.



**See Also**

Other branching: [tar\\_branch\\_names\\_raw\(\)](#), [tar\\_branch\\_names\(\)](#), [tar\\_branches\(\)](#), [tar\\_pattern\(\)](#)

**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(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	<i>Branch names</i>
------------------	---------------------

---

**Description**

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

**Usage**

```
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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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: [tar\\_branch\\_index\(\)](#), [tar\\_branch\\_names\\_raw\(\)](#), [tar\\_branches\(\)](#), [tar\\_pattern\(\)](#)

**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(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))
})
}
```

---

tar\_branch\_names\_raw *Branch names (raw version)*

---

**Description**

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

**Usage**

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

**Arguments**

name	Character of length 1, 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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: [tar\\_branch\\_index\(\)](#), [tar\\_branch\\_names\(\)](#), [tar\\_branches\(\)](#), [tar\\_pattern\(\)](#)

**Examples**

```

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	<i>List built targets.</i>
-----------	----------------------------

---

**Description**

List targets whose progress is "built".

**Usage**

```
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 <a href="#">all_of()</a> and <a href="#">starts_with()</a> .
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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**

```

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()
  })
}

```

---

tar\_cancel

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


---

**Description**

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

**Usage**

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

**Arguments**

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

**Details**

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

**See Also**

Other utilities: [tar\\_definition\(\)](#), [tar\\_envir\(\)](#), [tar\\_group\(\)](#), [tar\\_name\(\)](#), [tar\\_path\(\)](#), [tar\\_seed\(\)](#)

**Examples**

```

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.
  })
}

```

---

tar_canceled	<i>List canceled targets.</i>
--------------	-------------------------------

---

## 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 tidysselect helpers like <a href="#">all_of()</a> and <a href="#">starts_with()</a> .
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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)
  tar_make()
  tar_canceled()
  tar_canceled(starts_with("y_")) # see also all_of()
})
}
```

---

tar_condition	<i>Conditions</i>
---------------	-------------------

---

**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**

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

---

tar_config_get	<i>Get configuration settings.</i>
----------------	------------------------------------

---

**Description**

Read the custom settings for the current project in the optional YAML configuration file.

**Usage**

```
tar_config_get(
  name,
  config = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main")
)
```

**Arguments**

name	Character of length 1, name of the specific configuration setting to retrieve.
config	Character of length 1, file path of the YAML configuration file with targets project settings. The config argument specifies which YAML configuration file that tar_config_get() reads from or tar_config_set() writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always _targets.yaml unless you set another default path using the TAR_CONFIG environment variable, e.g. Sys.setenv(TAR_CONFIG = "custom.yaml"). This also has the effect of temporarily modifying the default arguments to other functions such as tar_make() because the default arguments to those functions are controlled by tar_config_get().
project	Character of length 1, name of the current targets project. Thanks to the config R package, targets YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The project argument allows you to set or get a configuration setting for a specific project for a given call to tar_config_set() or tar_config_get(). The default project is always called "main" unless you set another default project using the TAR_PROJECT environment variable, e.g. Sys.setenv(tar_project = "custom"). This also has the effect of temporarily modifying the default arguments to other functions such as tar_make() because the default arguments to those functions are controlled by tar_config_get().

**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.

**Configuration**

For several key functions like tar\_make(), the default values of arguments are controlled though tar\_config\_get(). tar\_config\_get() retrieves data from an optional YAML configuration file. You can control the settings in the YAML file programmatically with tar\_config\_set(). The default file path of this YAML file is \_targets.yaml, and you can set another path globally using the TAR\_CONFIG environment variable. The YAML file can store configuration settings for multiple projects, and you can globally set the default project with the TAR\_PROJECT environment variable. The structure of the YAML file follows rules similar to the config R package, e.g. projects can inherit settings from one another using the inherits field. Exceptions include:

1. There is no requirement to have a configuration named "default".

2. Other projects do not inherit from the default project automatically.
3. Not all fields need values because targets already has defaults.

targets does not actually invoke the config package. The implementation in targets was written from scratch without viewing or copying any part of the source code of config.

### See Also

Other configuration: [tar\\_config\\_set\(\)](#), [tar\\_config\\_unset\(\)](#), [tar\\_envvars\(\)](#), [tar\\_option\\_get\(\)](#), [tar\\_option\\_reset\(\)](#), [tar\\_option\\_set\(\)](#)

### 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)))
    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
  })
}
```

---

<code>tar_config_set</code>	<i>Set configuration settings.</i>
-----------------------------	------------------------------------

---

### Description

`tar_config_set()` writes special custom settings for the current project to an optional YAML configuration file.

### Usage

```
tar_config_set(
  inherits = NULL,
  reporter_make = NULL,
  reporter_outdated = NULL,
  store = NULL,
  shortcut = NULL,
  script = NULL,
  workers = NULL,
  config = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main")
)
```



**Arguments**

inherits	Character of length 1, name of the project from which the current project should inherit configuration settings. The current project is the project argument, which defaults to <code>Sys.getenv("TAR_PROJECT", "main")</code> . If the inherits argument NULL, the inherits setting is not modified. Use <code>tar_config_unset()</code> to delete a setting.
reporter_make	Character of length 1, reporter argument to <code>tar_make()</code> and related functions that run the pipeline. If the argument NULL, the setting is not modified. Use <code>tar_config_unset()</code> to delete a setting.
reporter_outdated	Character of length 1, reporter argument to <code>tar_outdated()</code> and related functions that do not run the pipeline. If the argument NULL, the setting is not modified. Use <code>tar_config_unset()</code> to delete a setting.
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: <code>_targets.yaml</code> ). Usually, the data store lives at <code>_targets</code> . Set store to a custom directory to specify a path other than <code>_targets/</code> . The path need not exist before the pipeline begins, and it need not end with <code>"_targets"</code> , but it must be writeable. For optimal performance, choose a storage location with fast read/write access. If the argument NULL, the setting is not modified. Use <code>tar_config_unset()</code> to delete a setting.
shortcut	logical of length 1, default shortcut argument to <code>tar_make()</code> and related functions. If the argument NULL, the setting is not modified. Use <code>tar_config_unset()</code> to delete a setting.
script	Character of length 1, path to the target script file that defines the pipeline ( <code>_targets.R</code> by default). This path should be either an absolute path or a path relative to the project root where you will call <code>tar_make()</code> and other functions. When <code>tar_make()</code> and friends run the script from the current working directory. If the argument NULL, the setting is not modified. Use <code>tar_config_unset()</code> to delete a setting.
workers	Positive numeric of length 1, workers argument of <code>tar_make_clustermq()</code> and related functions that run the pipeline with parallel computing among targets. If the argument NULL, the setting is not modified. Use <code>tar_config_unset()</code> to delete a setting.
config	Character of length 1, file path of the YAML configuration file with targets project settings. The config argument specifies which YAML configuration file that <code>tar_config_get()</code> reads from or <code>tar_config_set()</code> writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always <code>_targets.yaml</code> unless you set another default path using the <code>TAR_CONFIG</code> environment variable, e.g. <code>Sys.setenv(TAR_CONFIG = "custom.yaml")</code> . This also has the effect of temporarily modifying the default arguments to other functions such as <code>tar_make()</code> because the default arguments to those functions are controlled by <code>tar_config_get()</code> .
project	Character of length 1, name of the current targets project. Thanks to the config R package, targets YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The

project argument allows you to set or get a configuration setting for a specific project for a given call to `tar_config_set()` or `tar_config_get()`. The default project is always called "main" unless you set another default project using the `TAR_PROJECT` environment variable, e.g. `Sys.setenv(tar_project = "custom")`. This also has the effect of temporarily modifying the default arguments to other functions such as `tar_make()` because the default arguments to those functions are controlled by `tar_config_get()`.

## Value

NULL (invisibly)

## Configuration

For several key functions like `tar_make()`, the default values of arguments are controlled though `tar_config_get()`. `tar_config_get()` retrieves data from an optional YAML configuration file. You can control the settings in the YAML file programmatically with `tar_config_set()`. The default file path of this YAML file is `_targets.yaml`, and you can set another path globally using the `TAR_CONFIG` environment variable. The YAML file can store configuration settings for multiple projects, and you can globally set the default project with the `TAR_PROJECT` environment variable. The structure of the YAML file follows rules similar to the `config` R package, e.g. projects can inherit settings from one another using the `inherits` field. Exceptions include:

1. There is no requirement to have a configuration named "default".
2. Other projects do not inherit from the default project automatically.
3. Not all fields need values because `targets` already has defaults.

`targets` does not actually invoke the `config` package. The implementation in `targets` was written from scratch without viewing or copying any part of the source code of `config`.

## See Also

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

## 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)))
    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) # tar_read() knows about _targets.yaml too.
    file.exists("_targets") # FALSE
    file.exists(store_path) # TRUE
  })
}
```

---

tar_config_unset	<i>Unset configuration settings.</i>
------------------	--------------------------------------

---

## Description

Unset (i.e. delete) one or more custom settings for the current project from the optional YAML configuration file. After that, `tar_option_get()` will return the original default values for those settings for the project.

## Usage

```
tar_config_unset(
  names = character(),
  config = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main")
)
```

## Arguments

names	Character vector of configuration settings to delete from the current project.
config	Character of length 1, file path of the YAML configuration file with targets project settings. The config argument specifies which YAML configuration file that <code>tar_config_get()</code> reads from or <code>tar_config_set()</code> writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always <code>_targets.yaml</code> unless you set another default path using the <code>TAR_CONFIG</code> environment variable, e.g. <code>Sys.setenv(TAR_CONFIG = "custom.yaml")</code> . This also has the effect of temporarily modifying the default arguments to other functions such as <code>tar_make()</code> because the default arguments to those functions are controlled by <code>tar_config_get()</code> .
project	Character of length 1, name of the current targets project. Thanks to the <code>config R</code> package, targets YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The project argument allows you to set or get a configuration setting for a specific project for a given call to <code>tar_config_set()</code> or <code>tar_config_get()</code> . The default project is always called "main" unless you set another default project using the <code>TAR_PROJECT</code> environment variable, e.g. <code>Sys.setenv(tar_project = "custom")</code> . This also has the effect of temporarily modifying the default arguments to other functions such as <code>tar_make()</code> because the default arguments to those functions are controlled by <code>tar_config_get()</code> .

## Value

NULL (invisibly)

## Configuration

For several key functions like `tar_make()`, the default values of arguments are controlled through `tar_config_get()`. `tar_config_get()` retrieves data from an optional YAML configuration file. You can control the settings in the YAML file programmatically with `tar_config_set()`. The default file path of this YAML file is `_targets.yaml`, and you can set another path globally using the `TAR_CONFIG` environment variable. The YAML file can store configuration settings for multiple projects, and you can globally set the default project with the `TAR_PROJECT` environment variable. The structure of the YAML file follows rules similar to the `config` R package, e.g. projects can inherit settings from one another using the `inherits` field. Exceptions include:

1. There is no requirement to have a configuration named "default".
2. Other projects do not inherit from the default project automatically.
3. Not all fields need values because `targets` already has defaults.

`targets` does not actually invoke the `config` package. The implementation in `targets` was written from scratch without viewing or copying any part of the source code of `config`.

## See Also

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

## 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)))
    tar_config_get("store") # "_targets"
    store_path <- tempfile()
    tar_config_set(store = store_path)
    tar_config_get("store") # Shows a temp file.
    tar_config_unset("store")
    tar_config_get("store") # _targets
  })
}
```

---

tar\_cue

*Declare the rules that cue a target.*

---

## Description

Declare the rules that mark a target as outdated.

**Usage**

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

**Arguments**

mode	Cue mode. If "thorough", all the cues apply unless individually suppressed. If "always", then the target always runs. If "never", then the target does not run unless the metadata does not exist or the last run errored.
command	Logical, whether to rerun the target if command changed since last time.
depend	Logical, whether to rerun the target if the value of one of the dependencies changed.
format	Logical, whether to rerun the target if the user-specified storage format changed. The storage format is user-specified through <a href="#">tar_target()</a> or <a href="#">tar_option_set()</a> .
iteration	Logical, whether to rerun the target if the user-specified iteration method changed. The iteration method is user-specified through <a href="#">tar_target()</a> or <a href="#">tar_option_set()</a> .
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/outdated 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

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

---

tar\_definition

*For developers only: get the definition of the current target.*

---

### Description

For developers only: get the full definition of the target currently running. This target definition is the same kind of object produced by `tar_target()`.

**Usage**

```
tar_definition(
  default = targets::tar_target_raw("target_name", quote(identity()))
)
```

**Arguments**

**default** Environment, value to return if `tar_definition()` 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**

Most users should not use `tar_definition()` because accidental modifications could break the pipeline. `tar_definition()` only exists in order to support third-party interface packages, and even then the returned target definition is not modified..

**Value**

If called from a running target, `tar_definition()` returns the target object of the currently running target. See the "Target objects" section for details.

**Target objects**

Functions like `tar_target()` produce target objects, special objects with specialized sets of S3 classes. Target objects represent skippable steps of the analysis pipeline as described at <https://books.ropensci.org/targets/>. Please read the walkthrough at <https://books.ropensci.org/targets/walkthrough.html> to understand the role of target objects in analysis pipelines.

For developers, <https://wlandau.github.io/targetopia/contributing.html#target-factories> explains target factories (functions like this one which generate targets) and the design specification at <https://books.ropensci.org/targets-design/> details the structure and composition of target objects.

**See Also**

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

**Examples**

```
class(tar_definition())
tar_definition()$settings$name
if (identical(Sys.getenv("TAR_EXAMPLES"), "true")) {
  tar_dir({ # tar_dir() runs code from a temporary directory.
    tar_script(
      tar_target(x, tar_definition()$settings$memory, memory = "transient")
    )
  })
  tar_make(x)
  tar_read(x)
}
```

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**

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

**Arguments**

names	Names of the targets to remove from <code>_targets/objects/</code> . You can supply symbols or tidyselect helpers like <code>all_of()</code> and <code>starts_with()</code> .
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

**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**

```
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	<i>Code dependencies</i>
----------	--------------------------

---

## 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

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

---

tar_deps_raw	<i>Code dependencies (raw version)</i>
--------------	--

---

**Description**

Same as [tar\\_deps\(\)](#) except `expr` must already be an unquoted function or expression object.

**Usage**

```
tar_deps_raw(expr)
```

**Arguments**

`expr`            An R expression object or function.

**Value**

Character vector of the dependencies of a function or expression.

**See Also**

Other inspect: [tar\\_deps\(\)](#), [tar\\_glimpse\(\)](#), [tar\\_manifest\(\)](#), [tar\\_network\(\)](#), [tar\\_outdated\(\)](#), [tar\\_sitrep\(\)](#), [tar\\_validate\(\)](#), [tar\\_visnetwork\(\)](#)

**Examples**

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

---

tar_destroy	<i>Destroy all or part of the data store.</i>
-------------	---

---

**Description**

Destroy all or part of the data store written by [tar\\_make\(\)](#) and similar functions.

**Usage**

```
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: <ul style="list-style-type: none"> <li>• "all": destroy the entire data store (default: <code>_targets/</code>)</li> <li>• "meta": just delete the metadata file at <code>meta/meta</code> in the data store, which invalidates all the targets but keeps the data.</li> <li>• "process": just delete the progress data file at <code>meta/process</code> in the data store, which resets the metadata of the main process.</li> <li>• "progress": just delete the progress data file at <code>meta/progress</code> in the data store, which resets the progress tracking info.</li> <li>• "objects": delete all the target return values in <code>objects/</code> in the data store but keep progress and metadata. Dynamic files are not deleted this way.</li> <li>• "scratch": temporary files saved during <code>tar_make()</code> that should automatically get deleted except if R crashed.</li> <li>• "workspaces": compressed files in <code>workspaces/</code> in the data store with the saved workspaces of targets. See <code>tar_workspace()</code> for details.</li> </ul>
ask	Logical of length 1, whether to pause with a menu prompt before deleting files. To disable this menu, set the <code>TAR_ASK</code> environment variable to "false". <code>usethis::edit_r_environ()</code> can help set environment variables.
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

**Value**

Nothing.

**See Also**

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

**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() # Creates the _targets/ data store.
    tar_destroy()
  })
}
```

```
print(file.exists("_targets")) # Should be FALSE.
})
}
```

---

tar_dir	<i>Execute code in a temporary directory.</i>
---------	---

---

### Description

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

### Usage

```
tar_dir(code)
```

### Arguments

code	User-defined code.
------	--------------------

### Value

Return value of the user-defined code.

### See Also

Other utilities to extend targets: [tar\\_assert](#), [tar\\_condition](#), [tar\\_language](#), [tar\\_test\(\)](#)

### Examples

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

---

tar_edit	<i>Open the target script file for editing.</i>
----------	---

---

### 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()`

---

<code>tar_engine_knitr</code>	<i>Target Markdown knitr engine</i>
-------------------------------	-------------------------------------

---

**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 `tar_globals = TRUE` chunk).

### See Also

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

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

### Examples

```
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. Designed to be called while the target is running. The environment 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

Most users should not use `tar_envir()` because accidental modifications to `parent.env(tar_envir())` could break the pipeline. `tar_envir()` only exists in order to support third-party interface packages, and even then the returned environment is not modified.

## 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_definition()`, `tar_group()`, `tar_name()`, `tar_path()`, `tar_seed()`

## Examples

```

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)
  })
}

```

---

tar_envvars	<i>Show environment variables to customize targets</i>
-------------	--

---

### 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

```
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 `.Renvi`ron file so you do not lose your configuration when you restart your R session. Modify the project-level `.Renvi`ron file with `usethis::edit_r_envi`ron(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 `.Renvi`ron file.

### TAR\_CONFIG

The `TAR_CONFIG` environment variable controls the file path to the optional YAML configuration file with project settings. See the help file of `tar_config_set()` for details.

### TAR\_PROJECT

The `TAR_PROJECT` environment variable sets the name of project to set and get settings when working with the YAML configuration file. See the help file of `tar_config_set()` for details.



**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\\_config\\_unset\(\)](#), [tar\\_option\\_get\(\)](#), [tar\\_option\\_reset\(\)](#), [tar\\_option\\_set\(\)](#)

**Examples**

```
tar_envvars()
```

---

tar_errored	<i>List errored targets.</i>
-------------	------------------------------

---

**Description**

List targets whose progress is "errored".

**Usage**

```
tar_errored(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 <a href="#">all_of()</a> and <a href="#">starts_with()</a> .
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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\(\)](#), [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)
  tar_make()
  tar_errored()
  tar_errored(starts_with("y_")) # see also all_of()
})
}

```

---

tar_exist_meta	<i>Check if target metadata exists.</i>
----------------	---

---

**Description**

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

**Usage**

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

**Arguments**

store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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>.

**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**

```
tar_exist_meta()
```

---

tar_exist_objects	<i>Check if local output data exists for one or more targets.</i>
-------------------	---

---

### Description

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

### Usage

```
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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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>.

### 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

```
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>.

### 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>.

**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**

```
tar_exist_progress()
```

---

<code>tar_exist_script</code>	<i>Check if the target script file exists.</i>
-------------------------------	--

---

**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**

```
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()
```

---

tar_github_actions	<i>Set up GitHub Actions to run a targets pipeline</i>
--------------------	--

---

**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 <code>Sys.getenv("TAR_ASK")</code> . (Set to "true" or "false" with <code>Sys.setenv()</code> ). If ask and the TAR_ASK environment variable are both indeterminate, defaults to <code>interactive()</code> .

**Details**

Steps to set up continuous deployment:

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

**Value**

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

**See Also**

Other scripts: [tar\\_edit\(\)](#), [tar\\_helper\\_raw\(\)](#), [tar\\_helper\(\)](#), [tar\\_renv\(\)](#), [tar\\_script\(\)](#)

**Examples**

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

---

tar_glimpse	<i>Visualize an abridged fast dependency graph.</i>
-------------	---

---

**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**

```
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 <code>shortcut</code> is TRUE, all the targets upstream as well). Selecting a small subgraph using <code>names</code> could speed up the load time of the visualization. Unlike <code>allow</code> , <code>names</code> 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 <code>starts_with()</code> . 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 <code>names</code> argument. If <code>shortcut</code> is <code>FALSE</code> (default) then the function checks all targets upstream of <code>names</code> as far back as the dependency graph goes. If <code>TRUE</code> , then the function only checks the targets in <code>names</code> and uses stored metadata for information about upstream dependencies as needed. <code>shortcut = TRUE</code> 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, <code>shortcut = TRUE</code> only works if you set <code>names</code> .
allow	Optional, define the set of allowable vertices in the graph. Unlike <code>names</code> , <code>allow</code> is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to <code>NULL</code> to allow all vertices in the pipeline and environment (default). Otherwise, you can supply symbols or tidyselect helpers like <code>starts_with()</code> .
exclude	Optional, define the set of exclude vertices from the graph. Unlike <code>names</code> , <code>exclude</code> is invoked only after the graph is mostly resolved, so it will not speed up execution. Set to <code>NULL</code> to exclude no vertices. Otherwise, you can supply symbols or tidyselect helpers like <code>all_of()</code> and <code>starts_with()</code> .
level_separation	Numeric of length 1, <code>levelSeparation</code> argument of <code>visNetwork::visHierarchicalLayout()</code> . Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If <code>level_separation</code> is <code>NULL</code> , the <code>levelSeparation</code> argument of <code>visHierarchicalLayout()</code> defaults to 150.
degree_from	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. <code>degree_from</code> 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. <code>degree_to</code> controls the number of edges the neighborhood extends downstream.
callr_function	A function from <code>callr</code> to start a fresh clean R process to do the work. Set to <code>NULL</code> 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). <code>callr_function</code> needs to be <code>NULL</code> for interactive debugging, e.g. <code>tar_option_set(debug = "your_target")</code> . However, <code>callr_function</code> should not be <code>NULL</code> for serious reproducible work.
callr_arguments	A list of arguments to <code>callr_function</code> .
envir	An environment, where to run the target R script (default: <code>_targets.R</code> ) if <code>callr_function</code> is <code>NULL</code> . Ignored if <code>callr_function</code> is anything other than <code>NULL</code> . <code>callr_function</code> should only be <code>NULL</code> for debugging and testing purposes, not for serious runs of a pipeline, etc.  The <code>envir</code> argument of <code>tar_make()</code> and related functions always overrides the current value of <code>tar_option_get("envir")</code> in the current R session just before running the target script file, so whenever you need to set an alternative <code>envir</code> , you should always set it with <code>tar_option_set()</code> 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 <code>tar_config_get("script")</code> , which in turn defaults to <code>_targets.R</code> . When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <a href="#">tar_script()</a> , <a href="#">tar_config_get()</a> , and <a href="#">tar_config_set()</a> 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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

```
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` is a column. `tar_group` is a vector of positive integers from 1 to the number of groups. Rows with the same integer in `tar_group` belong to the same group, and branches are arranged in increasing order with respect to the integers in `tar_group`. The assignment of `tar_group` integers to group levels depends on the orderings inside the grouping variables and not the order of rows in the dataset. `dplyr::group_keys()` on the grouped data frame shows how the grouping variables correspond to the integers in the `tar_group` column.

**Value**

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

**See Also**

Other utilities: [tar\\_cancel\(\)](#), [tar\\_definition\(\)](#), [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)
          )
        )
      )
    })
  })
}
```

```

    ) %>%
      group_by(id) %>%
      tar_group(),
      iteration = "group"
    ),
    tar_target(
      sums,
      sum(data$x),
      pattern = map(data),
      iteration = "vector"
    )
  })
tar_make()
tar_read(sums) # Should be c(3, 7, 11).
})
}

```

---

tar\_helper

*Write a helper R script.*


---

### Description

Write a helper R script for a targets pipeline. Could be supporting functions or the 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, <code>tar_helper_raw()</code> creates it.
code	Quoted code to write to path. <code>tar_helper()</code> 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 <code>!!</code> and <code>!!!</code> . 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\\_raw\(\)](#), [tar\\_renv\(\)](#), [tar\\_script\(\)](#)

**Examples**

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

---

tar_helper_raw	<i>Write a helper R script (raw version).</i>
----------------	---

---

**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_raw(path = NULL, code = NULL)
```

**Arguments**

path	Character of length 1, path to write (or overwrite) code. If the parent directory does not exist, <code>tar_helper_raw()</code> creates it.
code	Expression object. <code>tar_helper_raw()</code> 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

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

---

tar_interactive	<i>Run if Target Markdown interactive mode is on.</i>
-----------------	---

---

## Description

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

## Usage

```
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](http://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

```
tar_interactive(message("In interactive mode."))
```

---

tar_invalidate	<i>Invalidate targets and global objects in the metadata.</i>
----------------	---

---

### 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 <code>all_of()</code> and <code>starts_with()</code> .
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

### 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

```
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.
})
}
```

---

tar_language	<i>Language</i>
--------------	-----------------

---

### Description

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

### Usage

```
tar_deparse_language(expr)

tar_deparse_safe(expr, collapse = "\n", backtick = TRUE)

tar_tidy_eval(expr, envir, tidy_eval)

tar_tidymselect_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 tidymselect 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_tidymselect_eval()` applies tidymselect selection with some special guardrails around NULL inputs.

### See Also

Other utilities to extend targets: [tar\\_assert](#), [tar\\_condition](#), [tar\\_dir\(\)](#), [tar\\_test\(\)](#)

### Examples

```
tar_deparse_language(quote(run_model()))
```

---

tar_load	<i>Load the values of targets.</i>
----------	------------------------------------

---

### 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 <code>all_of()</code> and <code>starts_with()</code> .
branches	Integer of indices of the branches to load for any targets that are patterns.
meta	Data frame of metadata from <code>tar_meta()</code> . <code>tar_read()</code> with the default arguments can be inefficient for large pipelines because all the metadata is stored in a single file. However, if you call <code>tar_meta()</code> beforehand and supply it to the meta argument, then successive calls to <code>tar_read()</code> may run much faster.
envir	Environment to put the loaded targets.
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

### 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\\_raw\(\)](#), [tar\\_meta\(\)](#), [tar\\_objects\(\)](#), [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(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()
})
}
```

---

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**

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

**Arguments**

envir	Environment to source the target script (default: <code>_targets.R</code> ). Defaults to the calling environment.
script	Character of length 1, path to the target script file that defines the pipeline ( <code>_targets.R</code> by default). This path should be either an absolute path or a path relative to the project root where you will call <a href="#">tar_make()</a> and other functions. When <a href="#">tar_make()</a> and friends run the script from the current working directory. If the argument <code>NULL</code> , the setting is not modified. Use <a href="#">tar_config_unset()</a> to delete a setting.

**Details**

This function first sources the target script file (default: `_targets.R`) to load 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**

```
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()))
})
}
```

---

tar\_load\_raw

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

---

**Description**

Same as [tar\\_load\(\)](#) except `names` is a character vector. Do not use in knitr or R Markdown reports with `tarchetypes::tar_knit()` or `tarchetypes::tar_render()`.

**Usage**

```
tar_load_raw(
  names,
  branches = NULL,
  meta = tar_meta(store = store),
```

```

  envir = parent.frame(),
  store = targets::tar_config_get("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 <code>tar_meta()</code> . <code>tar_read()</code> with the default arguments can be inefficient for large pipelines because all the metadata is stored in a single file. However, if you call <code>tar_meta()</code> beforehand and supply it to the meta argument, then successive calls to <code>tar_read()</code> may run much faster.
envir	Environment to put the loaded targets.
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

### 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

```

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
}

```

```

y2
})
}

```

---

tar\_make

*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

```

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 <code>all_of()</code> and <code>starts_with()</code> . 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 <code>shortcut</code> is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. <code>shortcut = TRUE</code> 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. <code>shortcut = TRUE</code> 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 <code>tar_config_get("reporter_make")</code> . Choices: <ul style="list-style-type: none"> <li>• "verbose": print one message for each target that runs (default).</li> <li>• "silent": print nothing.</li> <li>• "timestamp": print a time-stamped message for each target that runs.</li> </ul>  |

- "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()
```

```

    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	<i>Run a pipeline of targets in parallel with persistent clustermq workers.</i>
--------------------	---

---

### 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

```

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 <code>NULL</code> to check/build all the targets (default). Otherwise, you can supply symbols, a character vector, or <code>tidyselect</code> helpers like <code>all_of()</code> and <code>starts_with()</code> . 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: <ul style="list-style-type: none"> <li>• "verbose": print one message for each target that runs (default).</li> <li>• "silent": print nothing.</li> <li>• "timestamp": print a time-stamped message for each target that runs.</li> <li>• "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.</li> </ul>
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> # 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

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

---

tar\_make\_future

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

---

### Description

This function is like `tar_make()` except that targets run in parallel with transient future workers. It requires that you declare your `future::plan()` inside the 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 <code>all_of()</code> and <code>starts_with()</code> . 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 <code>shortcut</code> is FALSE (default) then the function checks all targets upstream of names as far back as the dependency graph goes. <code>shortcut = TRUE</code> 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. <code>shortcut = TRUE</code> 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 <code>tar_config_get("reporter_make")</code> . Choices: <ul style="list-style-type: none"> <li>• "verbose": print one message for each target that runs (default).</li> <li>• "silent": print nothing.</li> <li>• "timestamp": print a time-stamped message for each target that runs.</li> <li>• "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" <code>strptime()</code> format) of the last time the progress changed and printed to the screen.</li> </ul>
workers	Positive integer, maximum number of transient future workers allowed to run at any given time.
callr_function	A function from <code>callr</code> 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). <code>callr_function</code> needs to be NULL for interactive debugging, e.g. <code>tar_option_set(debug = "your_target")</code> . However, <code>callr_function</code> should not be NULL for serious reproducible work.
callr_arguments	A list of arguments to <code>callr_function</code> .

envir	<p>An environment, where to run the target R script (default: <code>_targets.R</code>) if <code>callr_function</code> is <code>NULL</code>. Ignored if <code>callr_function</code> is anything other than <code>NULL</code>. <code>callr_function</code> should only be <code>NULL</code> for debugging and testing purposes, not for serious runs of a pipeline, etc.</p> <p>The <code>envir</code> argument of <code>tar_make()</code> and related functions always overrides the current value of <code>tar_option_get("envir")</code> in the current R session just before running the target script file, so whenever you need to set an alternative <code>envir</code>, you should always set it with <code>tar_option_set()</code> from within the target script file. In other words, if you call <code>tar_option_set(envir = envir1)</code> in an interactive session and then <code>tar_make(envir = envir2, callr_function = NULL)</code>, then <code>envir2</code> will be used.</p>
script	<p>Character of length 1, path to the target script file. Defaults to <code>tar_config_get("script")</code>, which in turn defaults to <code>_targets.R</code>. When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <code>tar_script()</code>, <code>tar_config_get()</code>, and <code>tar_config_set()</code> for details about the target script file and how to set it persistently for a project.</p>
store	<p>Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code>, which in turn defaults to <code>_targets/</code>. When you set this argument, the value of <code>tar_config_get("store")</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.</p>

### 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

```
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()
})
}
```

---

tar_manifest	<i>Produce a data frame of information about your targets.</i>
--------------	--

---

### Description

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

### Usage

```
tar_manifest(
  names = NULL,
  fields = c("name", "command", "pattern"),
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function),
  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 tidysselect helpers like <code>all_of()</code> and <code>starts_with()</code> .
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 tidysselect helpers like <code>starts_with()</code> . 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 <code>tar_target()</code> , <code>tar_target_raw()</code> , or <code>tar_option_set()</code> . <ul style="list-style-type: none"> <li>• name: Name of the target.</li> <li>• command: the R command that runs when the target builds.</li> <li>• pattern: branching pattern of the target, if applicable.</li> <li>• format: Storage format.</li> <li>• iteration: Iteration mode for branching.</li> <li>• error: Error mode, what to do when the target fails.</li> <li>• memory: Memory mode, when to keep targets in memory.</li> <li>• storage: Storage mode for high-performance computing scenarios.</li> <li>• retrieval: Retrieval mode for high-performance computing scenarios.</li> <li>• deployment: Where/whether to deploy the target in high-performance computing scenarios.</li> <li>• 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 <code>tar_make_future()</code>).</li> </ul>

- resources: A list of target-specific resource requirements for `tar_make_future()`.
  - cue\_mode: Cue mode from `tar_cue()`.
  - cue\_depend: Depend cue from `tar_cue()`.
  - cue\_expr: Command cue from `tar_cue()`.
  - cue\_file: File cue from `tar_cue()`.
  - cue\_format: Format cue from `tar_cue()`.
  - cue\_iteration: Iteration cue from `tar_cue()`.
  - packages: List columns of packages loaded before building the target.
  - library: List column of library paths to load the packages.
- `callr_function` A function from `callr` to start a fresh clean R process to do the work. Set to `NULL` to run in the current session instead of an external process (but restart your R session just before you do in order to clear debris out of the global environment). `callr_function` needs to be `NULL` for interactive debugging, e.g. `tar_option_set(debug = "your_target")`. However, `callr_function` should not be `NULL` for serious reproducible work.
- `callr_arguments` A list of arguments to `callr_function`.
- `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.

## 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).

## See Also

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

## Examples

```

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

```

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

```

## Arguments

- |        |   |
|--------|---|
| names  | Optional, names of the targets. If supplied, <code>tar_meta()</code> only returns metadata on these targets. You can supply symbols or <code>tidyselect</code> helpers like <code>all_of()</code> and <code>starts_with()</code> . If NULL, all names are selected.   |
| fields | Optional, names of columns/fields to select. If supplied, <code>tar_meta()</code> only returns the selected metadata columns. If NULL, all fields are selected. You can supply symbols or <code>tidyselect</code> helpers like <code>all_of()</code> and <code>starts_with()</code> . The name column is always included first no matter what you select. Choices: <ul style="list-style-type: none"> <li>name: name of the target or global object.</li> </ul> |

- 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**

```
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	<i>Get the name of the target currently running.</i>
----------	--

---

**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_definition()`, `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: <ul style="list-style-type: none"> <li>• "silent": print nothing.</li> <li>• "forecast": print running totals of the checked and outdated targets found so far.</li> </ul>
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 <code>tar_config_get("script")</code> , which in turn defaults to <code>_targets.R</code> . When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <code>tar_script()</code> , <code>tar_config_get()</code> , and <code>tar_config_set()</code> 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

### 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

```
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)
  })
}
```

---

tar_newer	<i>List new targets</i>
-----------	-------------------------

---

### 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 <code>time = Sys.time - as.difftime(1, units = "weeks")</code> then <code>tar_newer()</code> 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 <code>all_of()</code> and <code>starts_with()</code> . 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

### 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()`

**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)))
    }, 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
    # so they run on the next tar_make().
    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**

```
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](http://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	<i>List saved targets</i>
-------------	---------------------------

---

**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 <a href="#">all_of()</a> or <a href="#">starts_with()</a> 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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()
  })
}
```

---

tar_older	<i>List old targets</i>
-----------	-------------------------

---

### Description

List all the targets whose last successful run occurred before a certain point in time. Combine with [tar\\_invalidate\(\)](#), you can use `tar_older()` to automatically rerun targets at regular intervals. See the examples for a demonstration.

### Usage

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

### Arguments

time	A POSIXct object of length 1, time threshold. Targets older than this time stamp are returned. For example, if <code>time = Sys.time() - as.difftime(1, units = "weeks")</code> then <code>tar_older()</code> returns targets older than one week ago.
names	Names of eligible targets. Targets excluded from names will not be returned even if they are old. You can supply symbols or tidyselect helpers like <a href="#">all_of()</a> and <a href="#">starts_with()</a> . 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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\\_newer\(\)](#), [tar\\_timestamp\\_raw\(\)](#), [tar\\_timestamp\(\)](#)

**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)))
    }, 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	<i>Get a target option.</i>
----------------	-----------------------------

---

**Description**

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

**Usage**

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

**Arguments**

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

**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\\_config\\_unset\(\)](#), [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	<i>Reset all target options.</i>
------------------	----------------------------------

---

**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\\_config\\_unset\(\)](#), [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	<i>Set target options.</i>
----------------	----------------------------

---

**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,
  enviro = NULL,
  format = NULL,
  iteration = NULL,
  error = NULL,
  memory = NULL,
  garbage_collection = NULL,
  deployment = NULL,
  priority = NULL,
  backoff = NULL,
  resources = NULL,
  storage = NULL,

```

```

retrieval = NULL,
cue = NULL,
debug = NULL,
workspaces = NULL,
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 <code>!!</code> to programmatically insert the values of global objects.
packages	Character vector of packages to load right before the target builds. Use <code>tar_option_set()</code> 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 <code>tar_option_set(imports = "yourAnalysisPackage")</code> early in your target script file (default: <code>_targets.R</code> ) then <code>tar_make()</code> will automatically rerun or skip targets in response to changes to the R functions and objects defined in <code>yourAnalysisPackage</code> . Does not account for low-level compiled code such as C/C++ or Fortran. If you supply multiple packages, e.g. <code>tar_option_set(imports = c("p1", "p2"))</code> , then the objects in <code>p1</code> override the objects in <code>p2</code> if there are name conflicts. Similarly, objects in <code>tar_option_get("envir")</code> override everything in <code>tar_option_get("imports")</code> .
library	Character vector of library paths to try when loading packages.
envir	<p>Environment containing functions and global objects common to all targets in the pipeline. The <code>envir</code> argument of <code>tar_make()</code> and related functions always overrides the current value of <code>tar_option_get("envir")</code> in the current R session just before running the target script file, so whenever you need to set an alternative <code>envir</code>, you should always set it with <code>tar_option_set()</code> from within the target script file. In other words, if you call <code>tar_option_set(envir = envir1)</code> in an interactive session and then <code>tar_make(envir = envir2, callr_function = NULL)</code>, then <code>envir2</code> will be used.</p> <p>If <code>envir</code> is the global environment, all the promise objects are diffused before sending the data to parallel workers in <code>tar_make_future()</code> and <code>tar_make_clustermq()</code>, but otherwise the environment is unmodified. This behavior improves performance by decreasing the size of data sent to workers.</p> <p>If <code>envir</code> 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 <code>envir</code>, targets attempts to remove potentially high memory objects that come directly from targets. That includes <code>tar_target()</code> objects of class <code>"tar_target"</code>, as well as objects of class <code>"tar_pipeline"</code> or <code>"tar_algorithm"</code>. This behavior improves performance by decreasing the size of data sent to workers.</p> <p>Package environments should not be assigned to <code>envir</code>. To include package objects as upstream dependencies in the pipeline, assign the package to the <code>packages</code> and <code>imports</code> arguments of <code>tar_option_set()</code>.</p>

format	Optional storage format for the target's return value. With the exception of <code>format = "file"</code> , each target gets a file in <code>_targets/objects</code> , 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: <ul style="list-style-type: none"> <li>"vector": branching happens with <code>vctrs::vec_slice()</code> and aggregation happens with <code>vctrs::vec_c()</code>.</li> <li>"list", branching happens with <code>[[]]</code> and aggregation happens with <code>list()</code>.</li> <li>"group": <code>dplyr::group_by()</code>-like functionality to branch over subsets of a data frame. The target's return value must be a data frame with a special <code>tar_group</code> 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 <code>tar_group()</code> function to see how you can create the special <code>tar_group</code> column with <code>dplyr::group_by()</code>.</li> </ul>
error	Character of length 1, what to do if the target stops and throws an error. Options: <ul style="list-style-type: none"> <li>"stop": the whole pipeline stops and throws an error.</li> <li>"continue": the whole pipeline keeps going.</li> <li>"abridge": any currently running targets keep running, but no new targets launch after that. (Visit <a href="https://books.ropensci.org/targets/debugging.html">https://books.ropensci.org/targets/debugging.html</a> to learn how to debug targets using saved workspaces.)</li> </ul>
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 <code>format = "aws_file"</code> , this memory strategy applies to temporary local copies of the file in <code>_targets/scratch/</code> : "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 <code>base::gc()</code> just before the target runs.
deployment	Character of length 1, only relevant to <code>tar_make_clustermq()</code> and <code>tar_make_future()</code> . 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 <code>tar_make_future()</code> ).
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. <code>tar_make_clustermq()</code> and <code>tar_make_future()</code> ) it can be expensive to repeatedly poll the priority queue if no targets are ready to process. The number of seconds between polls is <code>runif(1, 0.001, max(backoff, 0.001 * 1.5 ^ index))</code> , where <code>index</code> 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](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 <code>tar_resources()</code> with optional settings for high-performance computing functionality, alternative data storage formats, and other optional capabilities of targets. See <code>tar_resources()</code> for details.
storage	<p>Character of length 1, only relevant to <code>tar_make_clustermq()</code> and <code>tar_make_future()</code>. Must be one of the following values:</p> <ul style="list-style-type: none"> <li>• "main": the target's return value is sent back to the host machine and saved/uploaded locally.</li> <li>• "worker": the worker saves/uploads the value.</li> <li>• "none": almost never recommended. It is only for niche situations, e.g. the data needs to be loaded explicitly from another language. If you do use it, then the return value of the target is totally ignored when the target ends, but each downstream target still attempts to load the data file (except when <code>retrieval = "none"</code>).</li> </ul> <p>If you select <code>storage = "none"</code>, then the return value of the target's command is ignored, and the data is not saved automatically. As with dynamic files (<code>format = "file"</code> or <code>"aws_file"</code>) it is the responsibility of the user to write to <code>tar_path()</code> from inside the target. An example target could look something like <code>tar_target(x, saveRDS("value", tar_path(create_dir = TRUE))); "ignored", storage = "none")</code>.</p> <p>The distinguishing feature of <code>storage = "none"</code> (as opposed to <code>format = "file"</code> or <code>"aws_file"</code>) is that in the general case, downstream targets will automatically try to load the data from the data store as a dependency. As a corollary, <code>storage = "none"</code> is completely unnecessary if <code>format</code> is <code>"file"</code> or <code>"aws_file"</code>.</p>
retrieval	<p>Character of length 1, only relevant to <code>tar_make_clustermq()</code> and <code>tar_make_future()</code>. Must be one of the following values:</p> <ul style="list-style-type: none"> <li>• "main": the target's dependencies are loaded on the host machine and sent to the worker before the target builds.</li> <li>• "worker": the worker loads the targets dependencies.</li> <li>• "none": the dependencies are not loaded at all. This choice is almost never recommended. It is only for niche situations, e.g. the data needs to be loaded explicitly from another language.</li> </ul>
cue	An optional object from <code>tar_cue()</code> to customize the rules that decide whether the target is up to date.
debug	<p>Character vector of names of targets to run in debug mode. To use effectively, you must set <code>callr_function = NULL</code> and restart your R session just before running. You should also <code>tar_make()</code>, <code>tar_make_clustermq()</code>, or <code>tar_make_future()</code>. For any target mentioned in <code>debug</code>, targets will force the target to build locally (with <code>tar_cue(mode = "always")</code> and <code>deployment = "main"</code> in the settings) and pause in an interactive debugger to help you diagnose problems. This is like inserting a <code>browser()</code> statement at the beginning of the target's expression, but without invalidating any targets.</p>

`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_config_unset()`, `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_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\_outdated

*Check which targets are outdated.*

---

### Description

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

**Usage**

```
tar_outdated(
  names = NULL,
  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. <code>tar_outdated()</code> will check these targets and all upstream ancestors in the dependency graph. Set <code>names</code> to <code>NULL</code> to check/build all the targets (default). Otherwise, you can supply symbols or <code>tidyselect</code> helpers like <code>all_of()</code> and <code>starts_with()</code> . 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 <code>names</code> argument. If <code>shortcut</code> is <code>FALSE</code> (default) then the function checks all targets upstream of <code>names</code> as far back as the dependency graph goes. If <code>TRUE</code> , then the function only checks the targets in <code>names</code> and uses stored metadata for information about upstream dependencies as needed. <code>shortcut = TRUE</code> 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, <code>shortcut = TRUE</code> only works if you set <code>names</code> .
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: <code>_targets.R</code> ).
reporter	Character of length 1, name of the reporter to user. Controls how messages are printed as targets are checked. Choices: <ul style="list-style-type: none"> <li>• "silent": print nothing.</li> <li>• "forecast": print running totals of the checked and outdated targets found so far.</li> </ul>
callr_function	A function from <code>callr</code> to start a fresh clean R process to do the work. Set to <code>NULL</code> 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). <code>callr_function</code> needs to be <code>NULL</code> for interactive debugging, e.g. <code>tar_option_set(debug = "your_target")</code> . However, <code>callr_function</code> should not be <code>NULL</code> for serious reproducible work.

callr_arguments	A list of arguments to callr_function.
envir	An environment, where to run the target R script (default: <code>_targets.R</code> ) 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 <code>envir</code> argument of <code>tar_make()</code> and related functions always overrides the current value of <code>tar_option_get("envir")</code> in the current R session just before running the target script file, so whenever you need to set an alternative <code>envir</code> , you should always set it with <code>tar_option_set()</code> from within the target script file. In other words, if you call <code>tar_option_set(envir = envir1)</code> in an interactive session and then <code>tar_make(envir = envir2, callr_function = NULL)</code> , then <code>envir2</code> will be used.
script	Character of length 1, path to the target script file. Defaults to <code>tar_config_get("script")</code> , which in turn defaults to <code>_targets.R</code> . When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <code>tar_script()</code> , <code>tar_config_get()</code> , and <code>tar_config_set()</code> 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

### 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

```
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)
tar_outdated()
})
}

```

---

tar\_path

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


---

### Description

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

### Usage

```

tar_path(
  name = NULL,
  default = NA_character_,
  create_dir = FALSE,
  store = targets::tar_config_get("store")
)

```

### 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.
create_dir	Logical of length 1, whether to create dirname(tar_path()) in tar_path() itself. This is useful if you are writing to tar_path() from inside a storage = "none" target and need the parent directory of the file to exist.
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, file path of the return value of the target. If not called from inside a running target, 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.

For non-cloud storage formats, if you call tar\_path() with no arguments while target x is running, the name argument defaults to the name of the running target, so tar\_path() returns \_targets/objects/x.



For cloud-backed formats, `tar_path()` returns the path to the staging file in `_targets/scratch/`. That way, even if you select a cloud format (e.g. `tar_target(..., format = "aws_parquet", storage = "none")`) then you can still manually write to `tar_path(create_dir = TRUE)` and the `targets` package will automatically hash it and upload it to the AWS S3 bucket. This does not apply to formats `"file"` or `"aws_file"`, where you would never need `storage = "none"` anyway.

### See Also

Other utilities: `tar_cancel()`, `tar_definition()`, `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)
  })
}
```

---

tar_pattern	<i>Emulate dynamic branching.</i>
-------------	-----------------------------------

---

### Description

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

### Usage

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

### Arguments

pattern	Function call with the pattern specification.
...	Named integers, each of length 1. Each name is the name of a dependency target, and each integer is the length of the target (number of branches or slices). Names must be unique.
seed	Integer of length 1, random number generator seed to emulate the pattern reproducibly. (The <code>sample()</code> 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(
  tar_target(x, seq_len(2)),
  tar_target(y, head(letters, 2)),
  tar_target(dynamic, c(x, y), pattern = map(x, y)) # 2 branches
)
# Likewise for more complicated patterns.
list(
  tar_target(x, seq_len(2)),
  tar_target(y, head(letters, 2)),
  tar_target(z, head(LETTERS, 2)),
  tar_target(dynamic, c(x, y, z), pattern = cross(z, map(x, y))) #4 branches
)
# But you can emulate dynamic branching without running a pipeline
# in order to understand the patterns you are creating. Simply supply
# the pattern and the length of each dependency target.
# The returned data frame represents the branching structure of the pattern:
```

```

# One row per new branch, one column per dependency target, and
# one element per bud/branch in each dependency target.
tar_pattern(
  cross(x, map(y, z)),
  x = 2,
  y = 3,
  z = 3
)
tar_pattern(
  head(cross(x, map(y, z)), n = 2),
  x = 2,
  y = 3,
  z = 3
)

```

---

tar\_pid

*Get main process ID.*


---

### Description

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

### Usage

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

### Arguments

store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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 it with `ps::ps_is_running(ps::ps_handle(targets::tar_pid()))` before running another call to [tar\\_make\(\)](#) for the same project.

### Value

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

**See Also**

Other data: [tar\\_load\\_raw\(\)](#), [tar\\_load\(\)](#), [tar\\_meta\(\)](#), [tar\\_objects\(\)](#), [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, seq_len(2)),
        tar_target(y, 2 * x, pattern = map(x))
      )
    }, ask = FALSE)
  tar_make()
  Sys.getpid()
  tar_pid() # Different from the current PID.
})
}
```

---

tar\_poll

*Repeatedly poll progress in the R console.*


---

**Description**

Print the information in [tar\\_progress\\_summary\(\)](#) at regular intervals.

**Usage**

```
tar_poll(
  interval = 1,
  timeout = Inf,
  fields = c("skipped", "started", "built", "errored", "canceled", "since"),
  store = targets::tar_config_get("store")
)
```

**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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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**

```
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**

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

**Arguments**

names	Optional, names of the data points to return. If supplied, <code>tar_process()</code> returns only the rows of the names you select. You can supply symbols or <code>tidyselect</code> helpers like <a href="#">all_of()</a> and <a href="#">starts_with()</a> . If <code>NULL</code> , all names are selected.
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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**

```
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)
})
}
```

---

tar\_progress

*Read progress.*

---

**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

```
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**

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

**Arguments**

names	Optional, names of the targets. If supplied, <code>tar_progress()</code> only returns progress information on these targets. You can supply symbols or tidyselect helpers like <code>starts_with()</code> .
fields	Optional, names of progress data columns to read. Set to <code>NULL</code> to read all fields.
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

**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**

```
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()
})
}
```



---

tar\_progress\_summary *Summarize target progress.*

---

## Description

Summarize the progress of a run of the pipeline.

## Usage

```
tar_progress_summary(  
  fields = c("skipped", "started", "built", "errored", "canceled", "since"),  
  store = targets::tar_config_get("store")  
)
```

## 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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	<i>Remove targets that are no longer part of the pipeline.</i>
-----------	--

---

## Description

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

## Usage

```
tar_prune(
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function),
  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	<p>An environment, where to run the target R script (default: <code>_targets.R</code>) if <code>callr_function</code> is <code>NULL</code>. Ignored if <code>callr_function</code> is anything other than <code>NULL</code>. <code>callr_function</code> should only be <code>NULL</code> for debugging and testing purposes, not for serious runs of a pipeline, etc.</p> <p>The <code>envir</code> argument of <code>tar_make()</code> and related functions always overrides the current value of <code>tar_option_get("envir")</code> in the current R session just before running the target script file, so whenever you need to set an alternative <code>envir</code>, you should always set it with <code>tar_option_set()</code> from within the target script file. In other words, if you call <code>tar_option_set(envir = envir1)</code> in an interactive session and then <code>tar_make(envir = envir2, callr_function = NULL)</code>, then <code>envir2</code> will be used.</p>
script	<p>Character of length 1, path to the target script file. Defaults to <code>tar_config_get("script")</code>, which in turn defaults to <code>_targets.R</code>. When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <code>tar_script()</code>, <code>tar_config_get()</code>, and <code>tar_config_set()</code> for details about the target script file and how to set it persistently for a project.</p>
store	<p>Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code>, which in turn defaults to <code>_targets/</code>. When you set this argument, the value of <code>tar_config_get("store")</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.</p>

### 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 clean: `tar_delete()`, `tar_destroy()`, `tar_invalidate()`

### Examples

```
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()
  # Remove some targets from the pipeline.
  tar_script(list(tar_target(y1, 1 + 1)), ask = FALSE)
  # Keep only the remaining targets in the data store.
  tar_prune()
})
}
```

---

tar_read	<i>Read a target's value from storage.</i>
----------	--

---

### Description

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

### Usage

```
tar_read(
  name,
  branches = NULL,
  meta = tar_meta(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 <code>tar_meta()</code> . <code>tar_read()</code> with the default arguments can be inefficient for large pipelines because all the metadata is stored in a single file. However, if you call <code>tar_meta()</code> beforehand and supply it to the <code>meta</code> argument, then successive calls to <code>tar_read()</code> may run much faster.
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

### 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**

```

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)
  })
}

```

---

tar_read_raw	<i>Read a target's value from storage (raw version)</i>
--------------	---

---

**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 <code>tar_meta()</code> . <code>tar_read()</code> with the default arguments can be inefficient for large pipelines because all the metadata is stored in a single file. However, if you call <code>tar_meta()</code> beforehand and supply it to the meta argument, then successive calls to <code>tar_read()</code> may run much faster.
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

**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 renv can identify them automatically.

**Usage**

```
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.

<code>callr_function</code>	A function from <code>callr</code> to start a fresh clean R process to do the work. Set to <code>NULL</code> 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). <code>callr_function</code> needs to be <code>NULL</code> for interactive debugging, e.g. <code>tar_option_set(debug = "your_target")</code> . However, <code>callr_function</code> should not be <code>NULL</code> for serious reproducible work.
<code>callr_arguments</code>	A list of arguments to <code>callr_function</code> .
<code>envir</code>	An environment, where to run the target R script (default: <code>_targets.R</code> ) if <code>callr_function</code> is <code>NULL</code> . Ignored if <code>callr_function</code> is anything other than <code>NULL</code> . <code>callr_function</code> should only be <code>NULL</code> for debugging and testing purposes, not for serious runs of a pipeline, etc.  The <code>envir</code> argument of <code>tar_make()</code> and related functions always overrides the current value of <code>tar_option_get("envir")</code> in the current R session just before running the target script file, so whenever you need to set an alternative <code>envir</code> , you should always set it with <code>tar_option_set()</code> from within the target script file. In other words, if you call <code>tar_option_set(envir = envir1)</code> in an interactive session and then <code>tar_make(envir = envir2, callr_function = NULL)</code> , then <code>envir2</code> will be used.
<code>script</code>	Character of length 1, path to the target script file. Defaults to <code>tar_config_get("script")</code> , which in turn defaults to <code>_targets.R</code> . When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <code>tar_script()</code> , <code>tar_config_get()</code> , and <code>tar_config_set()</code> 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()
```

---

tar\_resources

*Target resources*

---

**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 <code>tar_resources_aws()</code> . 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 <code>tar_target()</code> .
clustermq	Output of function <code>tar_resources_clustermq()</code> . Optional clustermq settings for <code>tar_make_clustermq()</code> , including the <code>log_worker</code> and <code>template</code> arguments of <code>clustermq::workers()</code> .



feather	Output of function <code>tar_resources_feather()</code> . Non-default arguments to <code>arrow::read_feather()</code> and <code>arrow::write_feather()</code> for arrow/feather-based storage formats. Applies to all formats ending with the <code>"_feather"</code> suffix. For details on formats, see the format argument of <code>tar_target()</code> .
fst	Output of function <code>tar_resources_fst()</code> . Non-default arguments to <code>fst::read_fst()</code> and <code>fst::write_fst()</code> for fst-based storage formats. Applies to all formats ending with <code>"fst"</code> in the name. For details on formats, see the format argument of <code>tar_target()</code> .
future	Output of function <code>tar_resources_future()</code> . Optional future settings for <code>tar_make_future()</code> , including the resources argument of <code>future::future()</code> , which can include values to insert in template placeholders in <code>future.batchtools</code> template files. This is how to supply the resources argument of <code>future::future()</code> for targets. Resources supplied through <code>future::plan()</code> and <code>future::tweak()</code> are completely ignored.
parquet	Output of function <code>tar_resources_parquet()</code> . Non-default arguments to <code>arrow::read_parquet()</code> and <code>arrow::write_parquet()</code> for arrow/parquet-based storage formats. Applies to all formats ending with the <code>"_parquet"</code> suffix. For details on formats, see the format argument of <code>tar_target()</code> .
qs	Output of function <code>tar_resources_qs()</code> . Non-default arguments to <code>qs::qread()</code> and <code>qs::qsave()</code> for qs-based storage formats. Applies to all formats ending with the <code>"_qs"</code> suffix. For details on formats, see the format argument of <code>tar_target()</code> .
url	Output of function <code>tar_resources_url()</code> . Non-default settings for storage formats ending with the <code>"_url"</code> suffix. These settings include the <code>curl</code> handle for extra control over HTTP requests. For details on formats, see the format argument of <code>tar_target()</code> .

### 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_clustermq()`, `tar_resources_feather()`, `tar_resources_fst()`, `tar_resources_future()`, `tar_resources_parquet()`, `tar_resources_qs()`, `tar_resources_url()`

**Examples**

```
# 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))
  )
)
```

---

tar_resources_aws	<i>Target resources: AWS storage formats</i>
-------------------	--

---

**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**

```
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\\_clustermq\(\)](#), [tar\\_resources\\_feather\(\)](#), [tar\\_resources\\_fst\(\)](#), [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 = "aws_qs",
  resources = tar_resources(
    aws = tar_resources_aws(bucket = "yourbucketname"),
    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**

```
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**

```
# 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))
  )
)
```

---

tar\_resources\_feather *Target resources: feather storage formats*

---

**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**

```
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**

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

---

<code>tar_resources_fst</code>	<i>Target resources: fst storage formats</i>
--------------------------------	--

---

**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**

```
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 <code>future::plan()</code> object or NULL, a target-specific future plan.
resources	Named list, resources argument to <code>future::future()</code> . This argument is not supported in some versions of future. For versions of future where resources is not supported, instead supply resources to <code>future::tweak()</code> and assign the returned plan to the plan argument of <code>tar_resources_future()</code> .

**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**

```
# Somewhere in you 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**

```
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 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_qs()`, `tar_resources_url()`, `tar_resources()`

## Examples

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

---

tar_resources_qs	<i>Target resources: qs storage formats</i>
------------------	---

---

## 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

```
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

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

---

<code>tar_resources_url</code>	<i>Target resources: URL storage formats</i>
--------------------------------	--

---

## 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

```
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())
  )
)
}
```

---

tar\_script

*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

```
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 <code>library(targets)</code> line at the top of the target script file automatically (recommended). If TRUE, you do not need to explicitly put <code>library(targets)</code> in code.
ask	Logical, whether to ask before writing if the target script file already exists. If NULL, defaults to <code>Sys.getenv("TAR_ASK")</code> . (Set to "true" or "false" with <code>Sys.setenv()</code> ). If ask and the TAR_ASK environment variable are both indeterminate, defaults to <code>interactive()</code> .
script	Character of length 1, where to write the target script file. Defaults to <code>tar_config_get("script")</code> , which in turn defaults to <code>_targets.R</code> .

**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	<i>Get the random number generator seed of the target currently running.</i>
----------	--

---

## Description

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

## Usage

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

## Arguments

default	Integer, value to return if <code>tar_seed()</code> is called on its own outside a targets pipeline. Having a default lets users run things without <code>tar_make()</code> , 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\\_definition\(\)](#), [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)
```

```
  })
}
```

---

tar\_sitrep

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


---

## Description

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

## Usage

```
tar_sitrep(
  names = NULL,
  fields = NULL,
  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, <code>tar_sitrep()</code> only returns metadata on these targets. You can supply symbols or <code>tidyselect</code> helpers like <code>starts_with()</code> .   |
| fields | Optional, names of columns/fields to select. If supplied, <code>tar_sitrep()</code> only returns the selected metadata columns. You can supply symbols or <code>tidyselect</code> helpers like <code>all_of()</code> and <code>starts_with()</code> . The name column is always included first no matter what you select. Choices: <ul style="list-style-type: none"> <li>• name: name of the target or global object.</li> <li>• record: Whether the record cue is activated: TRUE if the target is not in the metadata (<code>tar_meta()</code>), or if the target errored during the last <code>tar_make()</code>, or if the class of the target changed.</li> <li>• always: Whether mode in <code>tar_cue()</code> is "always". If TRUE, <code>tar_make()</code> always runs the target.</li> <li>• never: Whether mode in <code>tar_cue()</code> is "never". If TRUE, <code>tar_make()</code> will only run if the record cue activates.</li> <li>• 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.</li> </ul> |

- `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 <code>tar_config_get("script")</code> , which in turn defaults to <code>_targets.R</code> . When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <code>tar_script()</code> , <code>tar_config_get()</code> , and <code>tar_config_set()</code> 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

## 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

```
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_sitrep()
  tar_meta(starts_with("y_")) # see also all_of()
})
}
```

---

tar_skipped	<i>List skipped targets.</i>
-------------	------------------------------

---

**Description**

List targets whose progress is "skipped".

**Usage**

```
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 <a href="#">all_of()</a> and <a href="#">starts_with()</a> .
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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**

```
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()
})
}
```



---

tar_started	<i>List started targets.</i>
-------------	------------------------------

---

### 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 <a href="#">all_of()</a> and <a href="#">starts_with()</a> .
store	Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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

```
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	<i>Declare a target.</i>
------------	--------------------------

---

## 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. <code>tar_target(downstream_target, f(upstream_target))</code> is a target named <code>downstream_target</code> which depends on a target <code>upstream_target</code> and a function <code>f()</code> . 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 <code>tar_meta(your_target, seed)</code> and run <code>set.seed()</code> 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 <code>x</code> and <code>y</code> , <code>tar_target(z, x + y, pattern = map(x, y))</code> implicitly defines branches of <code>z</code> that each compute <code>x[1] + y[1]</code> , <code>x[2] + y[2]</code> , and so on. See the user manual for details.
tidy_eval	Logical, whether to enable tidy evaluation when interpreting <code>command</code> and <code>pattern</code> . If <code>TRUE</code> , you can use the "bang-bang" operator <code>!!</code> to programmatically insert the values of global objects.
packages	Character vector of packages to load right before the target builds. Use <code>tar_option_set()</code> 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 <code>format = "file"</code> , each target gets a file in <code>_targets/objects</code> , 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: <ul style="list-style-type: none"> <li>"vector": branching happens with <code>vctrs::vec_slice()</code> and aggregation happens with <code>vctrs::vec_c()</code>.</li> <li>"list", branching happens with <code>[[]]</code> and aggregation happens with <code>list()</code>.</li> <li>"group": <code>dplyr::group_by()</code>-like functionality to branch over subsets of a data frame. The target's return value must be a data frame with a special <code>tar_group</code> 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 <code>tar_group()</code> function to see how you can create the special <code>tar_group</code> column with <code>dplyr::group_by()</code>.</li> </ul>
error	Character of length 1, what to do if the target stops and throws an error. Options: <ul style="list-style-type: none"> <li>"stop": the whole pipeline stops and throws an error.</li> <li>"continue": the whole pipeline keeps going.</li> <li>"abridge": any currently running targets keep running, but no new targets launch after that. (Visit <a href="https://books.ropensci.org/targets/debugging.html">https://books.ropensci.org/targets/debugging.html</a> to learn how to debug targets using saved workspaces.)</li> </ul>
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 <code>format = "aws_file"</code> , this memory strategy applies to temporary local copies of the file in <code>_targets/scratch/</code> : "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 <code>base::gc()</code> just before the target runs.
deployment	Character of length 1, only relevant to <code>tar_make_clustermq()</code> and <code>tar_make_future()</code> . 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 <code>tar_make_future()</code> ).
resources	Object returned by <code>tar_resources()</code> with optional settings for high-performance computing functionality, alternative data storage formats, and other optional capabilities of targets. See <code>tar_resources()</code> for details.
storage	Character of length 1, only relevant to <code>tar_make_clustermq()</code> and <code>tar_make_future()</code> . Must be one of the following values: <ul style="list-style-type: none"> <li>• "main": the target's return value is sent back to the host machine and saved/uploaded locally.</li> <li>• "worker": the worker saves/uploads the value.</li> <li>• "none": almost never recommended. It is only for niche situations, e.g. the data needs to be loaded explicitly from another language. If you do use it, then the return value of the target is totally ignored when the target ends, but each downstream target still attempts to load the data file (except when <code>retrieval = "none"</code>). If you select <code>storage = "none"</code>, then the return value of the target's command is ignored, and the data is not saved automatically. As with dynamic files (<code>format = "file"</code> or <code>"aws_file"</code>) it is the responsibility of the user to write to <code>tar_path()</code> from inside the target. An example target could look something like <code>tar_target(x, saveRDS("value", tar_path(create_dir = TRUE)); "ignored", storage = "none")</code>. The distinguishing feature of <code>storage = "none"</code> (as opposed to <code>format = "file"</code> or <code>"aws_file"</code>) is that in the general case, downstream targets will automatically try to load the data from the data store as a dependency. As a corollary, <code>storage = "none"</code> is completely unnecessary if <code>format</code> is <code>"file"</code> or <code>"aws_file"</code>.</li> </ul>
retrieval	Character of length 1, only relevant to <code>tar_make_clustermq()</code> and <code>tar_make_future()</code> . Must be one of the following values: <ul style="list-style-type: none"> <li>• "main": the target's dependencies are loaded on the host machine and sent to the worker before the target builds.</li> <li>• "worker": the worker loads the targets dependencies.</li> <li>• "none": the dependencies are not loaded at all. This choice is almost never recommended. It is only for niche situations, e.g. the data needs to be loaded explicitly from another language.</li> </ul>
cue	An optional object from <code>tar_cue()</code> 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`).

### Target objects

Functions like `tar_target()` produce target objects, special objects with specialized sets of S3 classes. Target objects represent skippable steps of the analysis pipeline as described at [https:](https://)

[//books.ropensci.org/targets/](https://books.ropensci.org/targets/). Please read the walkthrough at <https://books.ropensci.org/targets/walkthrough.html> to understand the role of target objects in analysis pipelines.

For developers, <https://wlandau.github.io/targetopia/contributing.html#target-factories> explains target factories (functions like this one which generate targets) and the design specification at <https://books.ropensci.org/targets-design/> details the structure and composition of target objects.

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

### 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	<i>Define a target using unrefined names and language objects.</i>
----------------	--

---

### 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. <code>tar_target(downstream_target, f(upstream_target))</code> is a target named <code>downstream_target</code> which depends on a target <code>upstream_target</code> and a function <code>f()</code> . 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 <code>tar_meta(your_target, seed)</code> and run <code>set.seed()</code> on the result to locally recreate the target's initial RNG state.
------	---

command	Similar to the <code>command</code> argument of <code>tar_target()</code> except the object must already be an expression instead of informally quoted code. <code>base::expression()</code> and <code>base::quote()</code> can produce such objects.
pattern	Similar to the <code>pattern</code> argument of <code>tar_target()</code> except the object must already be an expression instead of informally quoted code. <code>base::expression()</code> and <code>base::quote()</code> can produce such objects.
packages	Character vector of packages to load right before the target builds. Use <code>tar_option_set()</code> 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 <code>NULL</code> , 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 <code>string</code> to ignore changes in certain parts of the command. If <code>NULL</code> , the string is just deparsed from <code>command</code> (default).
format	Optional storage format for the target's return value. With the exception of <code>format = "file"</code> , each target gets a file in <code>_targets/objects</code> , 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: <ul style="list-style-type: none"> <li>• <code>"vector"</code>: branching happens with <code>vctrs::vec_slice()</code> and aggregation happens with <code>vctrs::vec_c()</code>.</li> <li>• <code>"list"</code>, branching happens with <code>[[]]</code> and aggregation happens with <code>list()</code>.</li> <li>• <code>"group"</code>: <code>dplyr::group_by()</code>-like functionality to branch over subsets of a data frame. The target's return value must be a data frame with a special <code>tar_group</code> 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 <code>tar_group()</code> function to see how you can create the special <code>tar_group</code> column with <code>dplyr::group_by()</code>.</li> </ul>
error	Character of length 1, what to do if the target stops and throws an error. Options: <ul style="list-style-type: none"> <li>• <code>"stop"</code>: the whole pipeline stops and throws an error.</li> <li>• <code>"continue"</code>: the whole pipeline keeps going.</li> <li>• <code>"abridge"</code>: any currently running targets keep running, but no new targets launch after that. (Visit <a href="https://books.ropensci.org/targets/debugging.html">https://books.ropensci.org/targets/debugging.html</a> to learn how to debug targets using saved workspaces.)</li> </ul>
memory	Character of length 1, memory strategy. If <code>"persistent"</code> , the target stays in memory until the end of the pipeline (unless <code>storage</code> is <code>"worker"</code> , in which case targets unloads the value from memory right after storing it in order to avoid sending copious data over a network). If <code>"transient"</code> , 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 <code>format = "aws_file"</code> , this memory strategy applies to temporary local copies of the file in <code>_targets/scratch/</code> : <code>"persistent"</code>



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 <code>base::gc()</code> just before the target runs.
deployment	Character of length 1, only relevant to <code>tar_make_clustermq()</code> and <code>tar_make_future()</code> . 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 <code>tar_make_future()</code> ).
resources	Object returned by <code>tar_resources()</code> with optional settings for high-performance computing functionality, alternative data storage formats, and other optional capabilities of targets. See <code>tar_resources()</code> for details.
storage	Character of length 1, only relevant to <code>tar_make_clustermq()</code> and <code>tar_make_future()</code> . Must be one of the following values: <ul style="list-style-type: none"> <li>• "main": the target's return value is sent back to the host machine and saved/uploaded locally.</li> <li>• "worker": the worker saves/uploads the value.</li> <li>• "none": almost never recommended. It is only for niche situations, e.g. the data needs to be loaded explicitly from another language. If you do use it, then the return value of the target is totally ignored when the target ends, but each downstream target still attempts to load the data file (except when <code>retrieval = "none"</code>).</li> </ul> <p>If you select <code>storage = "none"</code>, then the return value of the target's command is ignored, and the data is not saved automatically. As with dynamic files (<code>format = "file"</code> or <code>"aws_file"</code>) it is the responsibility of the user to write to <code>tar_path()</code> from inside the target. An example target could look something like <code>tar_target(x, saveRDS("value", tar_path(create_dir = TRUE))); "ignored", storage = "none")</code>.</p> <p>The distinguishing feature of <code>storage = "none"</code> (as opposed to <code>format = "file"</code> or <code>"aws_file"</code>) is that in the general case, downstream targets will automatically try to load the data from the data store as a dependency. As a corollary, <code>storage = "none"</code> is completely unnecessary if <code>format</code> is <code>"file"</code> or <code>"aws_file"</code>.</p>
retrieval	Character of length 1, only relevant to <code>tar_make_clustermq()</code> and <code>tar_make_future()</code> . Must be one of the following values: <ul style="list-style-type: none"> <li>• "main": the target's dependencies are loaded on the host machine and sent to the worker before the target builds.</li> <li>• "worker": the worker loads the targets dependencies.</li> <li>• "none": the dependencies are not loaded at all. This choice is almost never recommended. It is only for niche situations, e.g. the data needs to be loaded explicitly from another language.</li> </ul>
cue	An optional object from <code>tar_cue()</code> 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 the "Target objects" section for details.

**Target objects**

Functions like `tar_target()` produce target objects, special objects with specialized sets of S3 classes. Target objects represent skippable steps of the analysis pipeline as described at <https://books.ropensci.org/targets/>. Please read the walkthrough at <https://books.ropensci.org/targets/walkthrough.html> to understand the role of target objects in analysis pipelines.

For developers, <https://wlandau.github.io/targetopia/contributing.html#target-factories> explains target factories (functions like this one which generate targets) and the design specification at <https://books.ropensci.org/targets-design/> details the structure and composition of target objects.

**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**

```
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**

```
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	<i>Get the timestamp(s) of a target.</i>
---------------	--

---

**Description**

Get the timestamp associated with a target's last successful run.

**Usage**

```
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 <code>tar_timestamp()</code> 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

**Details**

`tar_timestamp()` checks the metadata in `_targets/meta/meta`, not the actual returned data of the target. The timestamp depends on the storage format of the target. If storage is local, e.g. formats like "rds" and "file", then the time stamp is the latest modification time of the target data files at the time the target last successfully ran. For non-local formats like "aws\_rds" and "url", then targets chooses instead to simply record the time the target last successfully ran.

**Value**

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

**See Also**

Other time: `tar_newer()`, `tar_older()`, `tar_timestamp_raw()`

**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))
    }, 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()
})
}

```

---

tar\_timestamp\_raw      *Get the timestamp(s) of a target (raw version).*

---

## Description

Get the time that a target last ran successfully.

## Usage

```

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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> for details about how to set the data store path persistently for a project.

## Details

`tar_timestamp_raw()` is like `tar_timestamp()` except it accepts a character string for name instead of a symbol. `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 `tar_timestamp_raw()` returns a POSIXct object at 1970-01-01 UTC.

**See Also**

Other time: `tar_newer()`, `tar_older()`, `tar_timestamp()`

**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))
    }, 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_make()
  })
}
```

---

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**

```
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](http://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**

```
tar_toggle(
  message("In interactive mode."),
  message("Not in interactive mode.")
)
```

---

tar_traceback	<i>Get a target's traceback</i>
---------------	---------------------------------

---

**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**

```
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: <code>_targets.R</code> ) to load user-defined global object dependencies into <code>envir</code> . If TRUE, then <code>envir</code> 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

```
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

```
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**

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

---

tar_validate	<i>Validate a pipeline of targets.</i>
--------------	--

---

**Description**

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

**Usage**

```
tar_validate(
  callr_function = callr::r,
  callr_arguments = targets::callr_args_default(callr_function),
  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	<p>An environment, where to run the target R script (default: <code>_targets.R</code>) if <code>callr_function</code> is <code>NULL</code>. Ignored if <code>callr_function</code> is anything other than <code>NULL</code>. <code>callr_function</code> should only be <code>NULL</code> for debugging and testing purposes, not for serious runs of a pipeline, etc.</p> <p>The <code>envir</code> argument of <code>tar_make()</code> and related functions always overrides the current value of <code>tar_option_get("envir")</code> in the current R session just before running the target script file, so whenever you need to set an alternative <code>envir</code>, you should always set it with <code>tar_option_set()</code> from within the target script file. In other words, if you call <code>tar_option_set(envir = envir1)</code> in an interactive session and then <code>tar_make(envir = envir2, callr_function = NULL)</code>, then <code>envir2</code> will be used.</p>
script	<p>Character of length 1, path to the target script file. Defaults to <code>tar_config_get("script")</code>, which in turn defaults to <code>_targets.R</code>. When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <code>tar_script()</code>, <code>tar_config_get()</code>, and <code>tar_config_set()</code> for details about the target script file and how to set it persistently for a project.</p>
store	<p>Character of length 1, path to the targets data store. Defaults to <code>tar_config_get("store")</code>, which in turn defaults to <code>_targets/</code>. When you set this argument, the value of <code>tar_config_get("store")</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.</p>

**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**

```
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**

```

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: <ul style="list-style-type: none"> <li>• "silent": print nothing.</li> <li>• "forecast": print running totals of the checked and outdated targets found so far.</li> </ul>
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 <code>tar_config_get("script")</code> , which in turn defaults to <code>_targets.R</code> . When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <a href="#">tar_script()</a> , <a href="#">tar_config_get()</a> , and <a href="#">tar_config_set()</a> 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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")) {
  tar_dir({ # tar_dir() runs code from a temporary directory.
    tar_script({
      tar_option_set()
      list(
        tar_target(y1, 1 + 1),
        tar_target(y2, 1 + 1),
        tar_target(z, y1 + y2)
      )
    })
  })
  tar_visnetwork()
  tar_visnetwork(allow = starts_with("y")) # see also all_of()
}
```

---

tar\_watch

*Shiny app to watch the dependency graph.*


---

**Description**

Launches a background process with a Shiny app that calls [tar\\_visnetwork\(\)](#) every few seconds. To embed this app in other apps, use the Shiny module in [tar\\_watch\\_ui\(\)](#) and [tar\\_watch\\_server\(\)](#).

**Usage**

```

tar_watch(
  seconds = 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 = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main"),
  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 <code>tar_visnetwork()</code> .

level_separation	Numeric of length 1, levelSeparation argument of <code>visNetwork::visHierarchicalLayout()</code> . Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If <code>level_separation</code> is NULL, the <code>levelSeparation</code> argument of <code>visHierarchicalLayout()</code> defaults to 150.
degree_from	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. <code>degree_from</code> 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. <code>degree_to</code> controls the number of edges the neighborhood extends downstream.
config	Character of length 1, file path of the YAML configuration file with targets project settings. The <code>config</code> argument specifies which YAML configuration file that <code>tar_config_get()</code> reads from or <code>tar_config_set()</code> writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always <code>_targets.yaml</code> unless you set another default path using the <code>TAR_CONFIG</code> environment variable, e.g. <code>Sys.setenv(TAR_CONFIG = "custom.yaml")</code> . This also has the effect of temporarily modifying the default arguments to other functions such as <code>tar_make()</code> because the default arguments to those functions are controlled by <code>tar_config_get()</code> .
project	Character of length 1, name of the current targets project. Thanks to the <code>config</code> R package, targets YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The <code>project</code> argument allows you to set or get a configuration setting for a specific project for a given call to <code>tar_config_set()</code> or <code>tar_config_get()</code> . The default project is always called "main" unless you set another default project using the <code>TAR_PROJECT</code> environment variable, e.g. <code>Sys.setenv(tar_project = "custom")</code> . This also has the effect of temporarily modifying the default arguments to other functions such as <code>tar_make()</code> because the default arguments to those functions are controlled by <code>tar_config_get()</code> .
height	Character of length 1, height of the <code>visNetwork</code> 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 <code>background</code> is TRUE.
host	Character of length 1, IPv4 address to listen on. Only relevant if <code>background</code> is TRUE.
port	Positive integer of length 1, TCP port to listen on. Only relevant if <code>background</code> is TRUE.
verbose	whether to print a spinner and informative messages. Only relevant if <code>background</code> 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 <code>--slave</code> 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 <code>print()</code> 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 <code>message()</code> 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 <code>stdout</code> , in which case they will be correctly interleaved. If this is the string <code>"2&gt;&amp;1"</code> , 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_errored()`, `tar_poll()`, `tar_progress_branches()`, `tar_progress_summary()`, `tar_progress()`, `tar_skipped()`, `tar_started()`, `tar_watch_server()`, `tar_watch_ui()`

### Examples

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



---

tar_watch_server	<i>Shiny module server for tar_watch()</i>
------------------	--

---

### Description

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

### Usage

```
tar_watch_server(
  id,
  height = "650px",
  exclude = ".Random.seed",
  config = Sys.getenv("TAR_CONFIG", "_targets.yaml"),
  project = Sys.getenv("TAR_PROJECT", "main")
)
```

### 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, file path of the YAML configuration file with targets project settings. The config argument specifies which YAML configuration file that <code>tar_config_get()</code> reads from or <code>tar_config_set()</code> writes to in a single function call. It does not globally change which configuration file is used in subsequent function calls. The default file path of the YAML file is always <code>_targets.yaml</code> unless you set another default path using the <code>TAR_CONFIG</code> environment variable, e.g. <code>Sys.setenv(TAR_CONFIG = "custom.yaml")</code> . This also has the effect of temporarily modifying the default arguments to other functions such as <code>tar_make()</code> because the default arguments to those functions are controlled by <code>tar_config_get()</code> .
project	Character of length 1, name of the current targets project. Thanks to the <code>config</code> R package, targets YAML configuration files can store multiple sets of configuration settings, with each set corresponding to its own project. The project argument allows you to set or get a configuration setting for a specific project for a given call to <code>tar_config_set()</code> or <code>tar_config_get()</code> . The default project is always called "main" unless you set another default project using the <code>TAR_PROJECT</code> environment variable, e.g. <code>Sys.setenv(tar_project = "custom")</code> . This also has the effect of temporarily modifying the default arguments to other functions such as <code>tar_make()</code> because the default arguments to those functions are controlled by <code>tar_config_get()</code> .

### 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\(\)](#)

---

tar_watch_ui	<i>Shiny module UI for tar_watch()</i>
--------------	--

---

**Description**

Use [tar\\_watch\\_ui\(\)](#) and [tar\\_watch\\_server\(\)](#) to include [tar\\_watch\(\)](#) as a Shiny module in an app.

**Usage**

```
tar_watch_ui(
  id,
  label = "tar_watch_label",
  seconds = 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	Character vector, label argument to <code>tar_visnetwork()</code> .
level_separation	Numeric of length 1, <code>levelSeparation</code> argument of <code>visNetwork::visHierarchicalLayout()</code> . Controls the distance between hierarchical levels. Consider changing the value if the aspect ratio of the graph is far from 1. If <code>level_separation</code> is NULL, the <code>levelSeparation</code> argument of <code>visHierarchicalLayout()</code> defaults to 150.
degree_from	Integer of length 1. When you click on a node, the graph highlights a neighborhood of that node. <code>degree_from</code> 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. <code>degree_to</code> controls the number of edges the neighborhood extends downstream.
height	Character of length 1, height of the <code>visNetwork</code> 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	<i>Load a saved workspace and seed for debugging.</i>
---------------	---

---

**Description**

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

**Usage**

```
tar_workspace(
  name,
  envir = parent.frame(),
  packages = TRUE,
  source = TRUE,
  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 <code>_targets.R</code> to load user-defined global object dependencies into <code>envir</code> . If <code>TRUE</code> , then <code>envir</code> 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 <code>tar_config_get("script")</code> , which in turn defaults to <code>_targets.R</code> . When you set this argument, the value of <code>tar_config_get("script")</code> is temporarily changed for the current function call. See <code>tar_script()</code> , <code>tar_config_get()</code> , and <code>tar_config_set()</code> 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</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.

**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**

```
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)
  # 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 <code>tar_config_get("store")</code> , which in turn defaults to <code>_targets/</code> . When you set this argument, the value of <code>tar_config_get("store")</code> is temporarily changed for the current function call. See <a href="#">tar_config_get()</a> and <a href="#">tar_config_set()</a> 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**

```
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**

```
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**

```
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, 29
  - tar\_interactive, 45
  - tar\_noninteractive, 68
  - tar\_toggle, 126
- \* **branching**
  - tar\_branch\_index, 8
  - tar\_branch\_names, 9
  - tar\_branch\_names\_raw, 10
  - tar\_branches, 7
  - tar\_pattern, 81
- \* **clean**
  - tar\_delete, 24
  - tar\_destroy, 26
  - tar\_invalidate, 46
  - tar\_prune, 90
- \* **configuration**
  - tar\_config\_get, 14
  - tar\_config\_set, 16
  - tar\_config\_unset, 19
  - tar\_envvars, 32
  - tar\_option\_get, 71
  - tar\_option\_reset, 72
  - tar\_option\_set, 73
- \* **data**
  - tar\_load, 48
  - tar\_load\_raw, 50
  - tar\_meta, 61
  - tar\_objects, 69
  - tar\_pid, 83
  - tar\_process, 85
  - tar\_read, 92
  - tar\_read\_raw, 93
- \* **debug**
  - tar\_load\_globals, 49
  - tar\_traceback, 127
  - tar\_workspace, 139
  - tar\_workspaces, 141
- \* **existence**
  - tar\_exist\_meta, 34
  - tar\_exist\_objects, 35
  - tar\_exist\_process, 36
  - tar\_exist\_progress, 36
  - tar\_exist\_script, 37
- \* **help**
  - targets-package, 4
  - use\_targets, 142
- \* **inspect**
  - tar\_deps, 25
  - tar\_deps\_raw, 26
  - tar\_glimpse, 39
  - tar\_manifest, 59
  - tar\_network, 64
  - tar\_outdated, 77
  - tar\_sitrep, 109
  - tar\_validate, 129
  - tar\_visnetwork, 130
- \* **pipeline**
  - tar\_make, 52
  - tar\_make\_clustermq, 54
  - tar\_make\_future, 56
- \* **progress**
  - tar\_built, 11
  - tar\_canceled, 13
  - tar\_errored, 33
  - tar\_poll, 84
  - tar\_progress, 86
  - tar\_progress\_branches, 87
  - tar\_progress\_summary, 89
  - tar\_skipped, 112
  - tar\_started, 113
  - tar\_watch, 133
  - tar\_watch\_server, 137
  - tar\_watch\_ui, 138
- \* **resources**
  - tar\_resources, 96
  - tar\_resources\_aws, 98
  - tar\_resources\_clustermq, 99



- tar\_resources\_feather, 100
- tar\_resources\_fst, 101
- tar\_resources\_future, 102
- tar\_resources\_parquet, 103
- tar\_resources\_qs, 104
- tar\_resources\_url, 105
- \* **scripts**
  - tar\_edit, 28
  - tar\_github\_actions, 38
  - tar\_helper, 43
  - tar\_helper\_raw, 44
  - tar\_renv, 94
  - tar\_script, 106
- \* **targets**
  - tar\_cue, 20
  - tar\_target, 114
  - tar\_target\_raw, 119
- \* **time**
  - tar\_newer, 67
  - tar\_older, 70
  - tar\_timestamp, 123
  - tar\_timestamp\_raw, 125
- \* **utilities to extend targets**
  - tar\_assert, 5
  - tar\_condition, 14
  - tar\_dir, 28
  - tar\_language, 47
  - tar\_test, 122
- \* **utilities**
  - tar\_cancel, 12
  - tar\_definition, 22
  - tar\_envir, 31
  - tar\_group, 41
  - tar\_name, 63
  - tar\_path, 80
  - tar\_seed, 108
- all\_of(), 11, 13, 24, 33, 40, 46, 48, 52, 54, 57, 59, 61, 65, 67, 69, 70, 78, 85, 86, 109, 112, 113, 131
- cross (tar\_pattern), 81
- head (tar\_pattern), 81
- library(), 94
- list(), 107, 116, 122
- map (tar\_pattern), 81
- sample (tar\_pattern), 81
- starts\_with(), 11, 13, 24, 33, 40, 46, 48, 52, 54, 57, 59, 61, 65, 67, 69, 70, 78, 85, 86, 88, 109, 112, 113, 131
- tail (tar\_pattern), 81
- tar\_assert, 5, 14, 28, 47, 123
- 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\_list (tar\_assert), 5
- tar\_assert\_match (tar\_assert), 5
- tar\_assert\_name (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, 82
- tar\_branch\_names, 8, 9, 9, 10, 82
- tar\_branch\_names(), 10
- tar\_branch\_names\_raw, 8–10, 10, 82

- tar\_branches, [7](#), [9](#), [10](#), [82](#)
- tar\_built, [11](#), [13](#), [33](#), [85](#), [87–89](#), [112](#), [113](#), [136](#), [138](#), [139](#)
- tar\_cancel, [12](#), [23](#), [31](#), [42](#), [64](#), [81](#), [108](#)
- tar\_cancel(), [89](#)
- tar\_canceled, [11](#), [13](#), [33](#), [85](#), [87–89](#), [112](#), [113](#), [136](#), [138](#), [139](#)
- tar\_condition, [7](#), [14](#), [28](#), [47](#), [123](#)
- tar\_config\_get, [14](#), [18](#), [20](#), [33](#), [72](#), [77](#)
- tar\_config\_get(), [7–11](#), [13](#), [24](#), [27](#), [29](#), [33–37](#), [41](#), [46](#), [48](#), [51](#), [53](#), [55](#), [56](#), [58](#), [60](#), [62](#), [66](#), [67](#), [69](#), [70](#), [79](#), [80](#), [83–85](#), [87–89](#), [91–93](#), [95](#), [111–113](#), [124](#), [125](#), [128–130](#), [133](#), [140](#), [141](#)
- tar\_config\_set, [16](#), [16](#), [20](#), [33](#), [72](#), [77](#)
- tar\_config\_set(), [7–11](#), [13](#), [24](#), [27](#), [29](#), [32–37](#), [41](#), [46](#), [48](#), [51](#), [53](#), [55](#), [56](#), [58](#), [60](#), [62](#), [66](#), [67](#), [69](#), [70](#), [79](#), [80](#), [83–85](#), [87–89](#), [91–93](#), [95](#), [111–113](#), [124](#), [125](#), [128–130](#), [133](#), [140](#), [141](#)
- tar\_config\_unset, [16](#), [18](#), [19](#), [33](#), [72](#), [77](#)
- tar\_config\_unset(), [17](#), [49](#)
- tar\_cue, [20](#), [118](#), [122](#)
- tar\_cue(), [60](#), [77](#), [109](#), [111](#)
- tar\_definition, [12](#), [22](#), [31](#), [42](#), [64](#), [81](#), [108](#)
- tar\_delete, [24](#), [27](#), [46](#), [91](#)
- tar\_deparse\_language (tar\_language), [47](#)
- tar\_deparse\_safe (tar\_language), [47](#)
- tar\_deps, [25](#), [26](#), [41](#), [60](#), [66](#), [79](#), [111](#), [130](#), [133](#)
- tar\_deps(), [22](#), [26](#)
- tar\_deps\_raw, [25](#), [26](#), [41](#), [60](#), [66](#), [79](#), [111](#), [130](#), [133](#)
- tar\_destroy, [24](#), [26](#), [46](#), [91](#)
- tar\_dir, [7](#), [14](#), [28](#), [47](#), [123](#)
- tar\_edit, [28](#), [39](#), [44](#), [96](#), [107](#)
- tar\_engine\_knitr, [29](#), [45](#), [68](#), [127](#)
- tar\_envir, [12](#), [23](#), [31](#), [42](#), [64](#), [81](#), [108](#)
- tar\_envvars, [16](#), [18](#), [20](#), [32](#), [72](#), [77](#)
- tar\_errored, [11](#), [13](#), [33](#), [85](#), [87–89](#), [112](#), [113](#), [136](#), [138](#), [139](#)
- tar\_exist\_meta, [34](#), [35–38](#)
- tar\_exist\_objects, [34](#), [35](#), [36–38](#)
- tar\_exist\_process, [34](#), [35](#), [36](#), [37](#), [38](#)
- tar\_exist\_progress, [34–36](#), [36](#), [38](#)
- tar\_exist\_script, [34–37](#), [37](#)
- tar\_github\_actions, [29](#), [38](#), [44](#), [96](#), [107](#)
- tar\_github\_actions(), [32](#)
- tar\_glimpse, [25](#), [26](#), [39](#), [60](#), [66](#), [79](#), [111](#), [130](#), [133](#)
- tar\_glimpse(), [59](#)
- tar\_group, [12](#), [23](#), [31](#), [41](#), [64](#), [81](#), [108](#)
- tar\_group(), [75](#), [115](#), [120](#)
- tar\_helper, [29](#), [39](#), [43](#), [44](#), [96](#), [107](#)
- tar\_helper(), [44](#)
- tar\_helper\_raw, [29](#), [39](#), [44](#), [44](#), [96](#), [107](#)
- tar\_interactive, [30](#), [45](#), [68](#), [127](#)
- tar\_invalidate, [24](#), [27](#), [46](#), [91](#)
- tar\_invalidate(), [70](#)
- tar\_language, [7](#), [14](#), [28](#), [47](#), [123](#)
- tar\_load, [48](#), [51](#), [63](#), [69](#), [84](#), [86](#), [92](#), [94](#)
- tar\_load(), [50](#)
- tar\_load\_globals, [49](#), [128](#), [141](#), [142](#)
- tar\_load\_raw, [49](#), [50](#), [63](#), [69](#), [84](#), [86](#), [92](#), [94](#)
- tar\_make, [52](#), [56](#), [58](#)
- tar\_make(), [15](#), [17–20](#), [23](#), [26](#), [27](#), [31](#), [38](#), [40](#), [49](#), [53–56](#), [58](#), [60](#), [63](#), [66](#), [74](#), [76](#), [77](#), [79](#), [80](#), [83](#), [85](#), [86](#), [90](#), [91](#), [95](#), [107–110](#), [118](#), [130](#), [132](#), [135](#), [137](#), [140](#)
- tar\_make\_clustermq, [53](#), [54](#), [58](#)
- tar\_make\_clustermq(), [17](#), [74–76](#), [90](#), [115](#), [116](#), [121](#)
- tar\_make\_future, [53](#), [56](#), [56](#)
- tar\_make\_future(), [59](#), [60](#), [74–76](#), [90](#), [115](#), [116](#), [121](#)
- tar\_manifest, [25](#), [26](#), [41](#), [59](#), [66](#), [79](#), [111](#), [130](#), [133](#)
- tar\_message\_run (tar\_condition), [14](#)
- tar\_meta, [49](#), [51](#), [61](#), [69](#), [84](#), [86](#), [92](#), [94](#)
- tar\_meta(), [48](#), [51](#), [63](#), [92](#), [93](#), [109](#)
- tar\_name, [12](#), [23](#), [31](#), [42](#), [63](#), [81](#), [108](#)
- tar\_network, [25](#), [26](#), [41](#), [60](#), [64](#), [79](#), [111](#), [130](#), [133](#)
- tar\_newer, [67](#), [70](#), [124](#), [126](#)
- tar\_noninteractive, [30](#), [45](#), [68](#), [127](#)
- tar\_objects, [49](#), [51](#), [63](#), [69](#), [84](#), [86](#), [92](#), [94](#)
- tar\_older, [67](#), [70](#), [124](#), [126](#)
- tar\_option\_get, [16](#), [18](#), [20](#), [33](#), [71](#), [72](#), [77](#)
- tar\_option\_get(), [19](#), [73](#)
- tar\_option\_reset, [16](#), [18](#), [20](#), [33](#), [72](#), [72](#), [77](#)
- tar\_option\_set, [16](#), [18](#), [20](#), [33](#), [72](#), [73](#)
- tar\_option\_set(), [21](#), [59](#), [71](#), [72](#), [95–101](#), [103–107](#), [140](#)
- tar\_outdated, [25](#), [26](#), [41](#), [60](#), [66](#), [77](#), [111](#), [130](#), [133](#)
- tar\_outdated(), [17](#)

- tar\_path, [12](#), [23](#), [31](#), [42](#), [64](#), [80](#), [108](#)
- tar\_path(), [76](#), [116](#), [121](#)
- tar\_pattern, [8–10](#), [81](#)
- tar\_pid, [49](#), [51](#), [63](#), [69](#), [83](#), [86](#), [92](#), [94](#)
- tar\_poll, [11](#), [13](#), [33](#), [84](#), [87–89](#), [112](#), [113](#),  
[136](#), [138](#), [139](#)
- tar\_process, [49](#), [51](#), [63](#), [69](#), [84](#), [85](#), [92](#), [94](#)
- tar\_progress, [11](#), [13](#), [33](#), [85](#), [86](#), [88](#), [89](#), [112](#),  
[113](#), [136](#), [138](#), [139](#)
- tar\_progress(), [63](#)
- tar\_progress\_branches, [11](#), [13](#), [33](#), [85](#), [87](#),  
[87](#), [89](#), [112](#), [113](#), [136](#), [138](#), [139](#)
- tar\_progress\_summary, [11](#), [13](#), [33](#), [85](#), [87](#),  
[88](#), [89](#), [112](#), [113](#), [136](#), [138](#), [139](#)
- tar\_progress\_summary(), [84](#)
- tar\_prune, [24](#), [27](#), [46](#), [90](#)
- tar\_read, [49](#), [51](#), [63](#), [69](#), [84](#), [86](#), [92](#), [94](#)
- tar\_read(), [93](#), [118](#)
- tar\_read\_raw, [49](#), [51](#), [63](#), [69](#), [84](#), [86](#), [92](#), [93](#)
- tar\_renv, [29](#), [39](#), [44](#), [94](#), [107](#)
- tar\_resources, [96](#), [99–106](#)
- tar\_resources\_aws, [97](#), [98](#), [100–106](#)
- tar\_resources\_clustermq, [97](#), [99](#), [99](#),  
[101–106](#)
- tar\_resources\_feather, [97](#), [99](#), [100](#), [100](#),  
[102–106](#)
- tar\_resources\_fst, [97](#), [99–101](#), [101](#),  
[103–106](#)
- tar\_resources\_future, [97](#), [99–102](#), [102](#),  
[104–106](#)
- tar\_resources\_parquet, [97](#), [99–103](#), [103](#),  
[105](#), [106](#)
- tar\_resources\_qs, [97](#), [99–104](#), [104](#), [106](#)
- tar\_resources\_url, [97](#), [99–105](#), [105](#)
- tar\_script, [29](#), [39](#), [44](#), [96](#), [106](#)
- tar\_script(), [29](#), [32](#), [37](#), [41](#), [43](#), [44](#), [53](#), [55](#),  
[58](#), [60](#), [66](#), [79](#), [91](#), [95](#), [111](#), [129](#), [130](#),  
[133](#), [140](#)
- tar\_seed, [12](#), [23](#), [31](#), [42](#), [64](#), [81](#), [108](#)
- tar\_sitrep, [25](#), [26](#), [41](#), [60](#), [66](#), [79](#), [109](#), [130](#),  
[133](#)
- tar\_skipped, [11](#), [13](#), [33](#), [85](#), [87–89](#), [112](#), [113](#),  
[136](#), [138](#), [139](#)
- tar\_started, [11](#), [13](#), [33](#), [85](#), [87–89](#), [112](#), [113](#),  
[136](#), [138](#), [139](#)
- tar\_target, [22](#), [114](#), [122](#)
- tar\_target(), [21](#), [22](#), [59](#), [62](#), [71–73](#), [81](#), [82](#),  
[95–101](#), [103–107](#), [119](#), [120](#)
- tar\_target\_raw, [22](#), [118](#), [119](#)
- tar\_target\_raw(), [59](#), [71–73](#)
- tar\_test, [7](#), [14](#), [28](#), [47](#), [122](#)
- tar\_throw\_file(tar\_condition), [14](#)
- tar\_throw\_run(tar\_condition), [14](#)
- tar\_throw\_validate(tar\_condition), [14](#)
- tar\_tidy\_eval(tar\_language), [47](#)
- tar\_tidyselect\_eval(tar\_language), [47](#)
- tar\_timestamp, [67](#), [70](#), [123](#), [126](#)
- tar\_timestamp\_raw, [67](#), [70](#), [124](#), [125](#)
- tar\_toggle, [30](#), [45](#), [68](#), [126](#)
- tar\_traceback, [50](#), [127](#), [141](#), [142](#)
- tar\_unscript, [128](#)
- tar\_unscript(), [128](#)
- tar\_validate, [25](#), [26](#), [41](#), [60](#), [66](#), [79](#), [111](#),  
[129](#), [133](#)
- tar\_visnetwork, [25](#), [26](#), [41](#), [60](#), [66](#), [79](#), [111](#),  
[130](#), [130](#)
- tar\_visnetwork(), [39](#), [59](#), [133](#), [134](#), [136](#), [139](#)
- tar\_warn\_deprecate(tar\_condition), [14](#)
- tar\_warn\_run(tar\_condition), [14](#)
- tar\_warn\_validate(tar\_condition), [14](#)
- tar\_watch, [11](#), [13](#), [33](#), [85](#), [87–89](#), [112](#), [113](#),  
[133](#), [138](#), [139](#)
- tar\_watch(), [137](#), [138](#)
- tar\_watch\_server, [11](#), [13](#), [33](#), [85](#), [87–89](#),  
[112](#), [113](#), [136](#), [137](#), [139](#)
- tar\_watch\_server(), [133](#), [138](#)
- tar\_watch\_ui, [11](#), [13](#), [33](#), [85](#), [87–89](#), [112](#),  
[113](#), [136](#), [138](#), [138](#)
- tar\_watch\_ui(), [133](#), [137](#)
- tar\_workspace, [50](#), [128](#), [139](#), [142](#)
- tar\_workspace(), [27](#), [77](#), [127](#), [141](#), [142](#)
- tar\_workspaces, [50](#), [128](#), [141](#), [141](#)
- targets-package, [4](#)
- use\_targets, [4](#), [142](#)