

Package: tarpolyglot (via r-universe)

September 22, 2026

Type Package

Title Run 'Python', 'Julia', 'Rust', and 'C++' Inside 'targets'
Pipeline Steps

Version 0.3.0

Description Adds target constructors that make it easy to use 'Python', 'Julia', 'Rust', and 'C++' inside a 'targets' pipeline using 'reticulate', 'JuliaCall', 'rextendr', and 'Rcpp'. Provides `tar_target_py()`, `tar_target_jl()`, `tar_target_rs()`, and `tar_target_cpp()` (with matching `_raw()` variants), each mirroring 'targets' `tar_target()` and `tar_target_raw()` functions. 'Python' and 'Julia' steps run a script via a live interpreter with optional R pre- and post-scripts; 'Rust' and 'C++' steps compile functions and call them from an R post-script. Results are then returned either as converted R objects or as files written to disk (format = ``file"). Dynamic branching, environment/version selection, a 'crew' controller for isolation, and the full set of `tar_target_raw()` arguments are supported.

License MIT + file LICENSE

URL <https://github.com/Pierre9344/tarpolyglot>,
<https://pierre9344.github.io/tarpolyglot/>

BugReports <https://github.com/Pierre9344/tarpolyglot/issues>

Encoding UTF-8

Roxygen list(markdown = TRUE)

Imports targets, reticulate, JuliaCall, crew, rextendr, Rcpp, callr,
cli

Suggests knitr, rmarkdown, tarchetypes, testthat (>= 3.2.0), withr,
RcppArmadillo

Config/testthat/edition 3

VignetteBuilder knitr

SystemRequirements The external language toolchains are optional and each is only required by its matching targets constructor: a Python installation for `tar_target_py()`, a Julia installation for `tar_target_jl()`, a Rust toolchain with Cargo for `tar_target_rs()` (on Windows, the GNU toolchain, to match R's ABI), and a C++ compiler reachable by R (e.g. via Rtools on Windows) for `tar_target_cpp()`. A pipeline that uses a single language needs only that language's toolchain.

Config/roxygen2/version 8.1.0

Config/pak/sysreqs libglpk-dev libicu-dev libpng-dev libxml2-dev python3 libclang-dev

Repository <https://pierre9344.r-universe.dev>

Date/Publication 2026-09-03 05:01:34 UTC

RemoteUrl <https://github.com/pierre9344/tarpolyglot>

RemoteRef HEAD

RemoteSha 50131042a9d0d9ad45bd5331202f8f78dee8d507

Contents

<code>polyglot_controller</code>	3
<code>run_cpp_step</code>	4
<code>run_jl_step</code>	6
<code>run_py_step</code>	9
<code>run_rs_step</code>	11
<code>tar_code</code>	13
<code>tar_polyglot_log</code>	15
<code>tar_target_cpp</code>	16
<code>tar_target_cpp_raw</code>	20
<code>tar_target_jl</code>	23
<code>tar_target_jl_raw</code>	27
<code>tar_target_path</code>	31
<code>tar_target_py</code>	32
<code>tar_target_py_raw</code>	36
<code>tar_target_rs</code>	41
<code>tar_target_rs_raw</code>	45
<code>tarpolyglot_cross</code>	48
<code>tarpolyglot_head</code>	49
<code>tarpolyglot_map</code>	50
<code>tarpolyglot_sample</code>	52
<code>tarpolyglot_slice</code>	53
<code>tarpolyglot_tail</code>	54
<code>toolchain_check</code>	56

Index

58

`polyglot_controller` *A crew controller preconfigured for polyglot isolation*

Description

Returns a `crew::crew_controller_local()` tuned for pipelines that run Python or Julia steps. Because reticulate and JuliaCall embed a **single interpreter per process** (see `vignette("get_started", package = "tarpolyglot")`), the only way to get a fresh interpreter per target, and to use different environments/versions across targets, is process-level isolation. Passing this controller to `targets::tar_option_set()` sends targets to separate worker processes; with `tasks_max = 1` each worker is retired after one task, so every foreign step gets a clean interpreter that is torn down when the task finishes.

Usage

```
polyglot_controller(
  workers = 2L,
  tasks_max = 1L,
  seconds_idle = 30,
  log = NULL,
  ...
)
```

Arguments

<code>workers</code>	Integer number of parallel worker processes. Default 2.
<code>tasks_max</code>	Integer max tasks a worker runs before it is retired (and its embedded interpreter torn down). Default 1 for maximum isolation: a fresh Python/Julia per target. Raise it (or set Inf) to reuse interpreters for throughput, accepting that same-language steps on a worker then share the foreign global namespace and one environment.
<code>seconds_idle</code>	Numeric seconds an idle worker waits before shutting down. Default 30.
<code>log</code>	Optional <code>tar_polyglot_log()</code> object turning on per-step stdout/stderr log files for Python and Julia steps (see that function; Rust is not covered). Default NULL (no logging, the previous behavior).
<code>...</code>	Further arguments passed to <code>crew::crew_controller_local()</code> , e.g. <code>seconds_timeout</code> , <code>garbage_collection</code> , or <code>reset_globals</code> .

Details

This is the recommended default for polyglot pipelines. Put it near the top of `_targets.R`:

```
targets::tar_option_set(controller = tarpolyglot::polyglot_controller())
```

Value

A crew controller object, ready for `targets::tar_option_set(controller = ...)`.

See Also

[tar_target_py\(\)](#), [tar_target_jl\(\)](#), [tar_polyglot_log\(\)](#)

Examples

```
# Constructing a controller starts no worker process: targets launches them
# when the pipeline runs, so this example needs no Python.
controller <- polyglot_controller(workers = 4L)
inherits(controller, "crew_class_controller")

# In _targets.R you hand it to targets::tar_option_set(), alongside the
# steps that will use it. The option is reset here so the example leaves
# no global state behind.
targets::tar_option_set(controller = controller)
list(
  tar_target_py(x, "scripts/step.py", retrieve = "result")
)
targets::tar_option_reset()
```

run_cpp_step

Execute a C++ step (worker behind tar_target_cpp)

Description

Compiles the `// [[Rcpp::export]]` functions in a C++ script with `Rcpp::sourceCpp()`, exposing them as R functions in a fresh environment, then evaluates an R **post-script** in that environment where you call those functions and return the result. Upstream inputs are bound in the same environment. This is the function the target built by [tar_target_cpp\(\)](#) calls; it is exported so the call resolves at run time, but package users should not call it directly.

Usage

```
run_cpp_step(
  script,
  post_script = NULL,
  inputs = list(),
  output = "object",
  files = NULL,
  depends = NULL,
  name = NULL
)
```

Arguments

script	Path to the C++ script containing <code>// [[Rcpp::export]]</code> functions (required). Accepts a file path or an inline tar_code() carrier; see the "Script arguments" section below.
--------	---

post_script	Path to an R script evaluated after compilation. The compiled C++ functions and the named inputs are in scope; its last expression is the target value (object mode), or it returns a character vector of file paths (file mode). Required for object mode. Accepts a file path or an inline <code>tar_code()</code> carrier; see the "Script arguments" section below.
inputs	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").
files	Optional character vector of file paths to return when no post_script is supplied in file mode.
depends	Optional character vector of extension packages (e.g. <code>c("RcppArmadillo", "RcppEigen")</code>), passed straight through as <code>// [[Rcpp::depends(...)]]</code> would be. Usually unnecessary: declaring <code>// [[Rcpp::depends(pkgname)]]</code> directly in the C++ source (the normal Rcpp convention) already works, so this argument is only useful if you would rather not repeat that in the source itself.
name	Character string, the step's target name. Supplied automatically by the constructor; used only to name this step's log files when <code>polyglot_controller()</code> was given a <code>tar_polyglot_log()</code> (NULL – the default – disables logging for a direct call). Only <code>Rcpp::Rcout/Rcpp::Rcerr</code> output from the post-script's calls into compiled code is captured; see <code>tar_polyglot_log()</code> .

Details

Unlike Python/Julia there is **no pre-script** for C++ and no live interpreter: `sourceCpp()` compiles a shared library and R calls into it with real type conversion (via **Rcpp**). Header-only extension packages such as `RcppArmadillo/RcppEigen` need no special handling: declare them with a `// [[Rcpp::depends(pkgname)]]` attribute directly in the C++ source, exactly as in any other Rcpp usage, and `sourceCpp()` picks it up. `sourceCpp()` compiles via R's own configured toolchain (R CMD SHLIB), so, unlike Rust, there is no separate compiler/ABI to match – on Windows this function still puts Rtools on PATH for the build itself, since a bare or crew-worker process may not otherwise have it there.

Value

The value of the post-script (object mode) or a character vector of normalised file paths (file mode).

Script arguments

A worker receives whatever the constructor already resolved, which is one of two things: a **path to a file** on disk, or an **inline carrier** built by `tar_code()` that holds the code in memory. Both are accepted, so a direct call may pass either.

`tar_target_path()` is deliberately *not* a third form at this level. It is a constructor-level convenience: `tar_target_py()` and the other constructors rewrite it while the pipeline's DAG is built,

so that by the time a worker runs it has already become the ordinary file path held by the upstream target. Handing the result of `tar_target_path()` straight to a worker therefore does not resolve to a file. The three forms as written in `_targets.R`, and which of them tracks your edits, are covered in vignette("scripts").

See Also

`tar_target_cpp()`, `run_py_step()`, `run_jl_step()`, `run_rs_step()`

Examples

```
# This worker compiles C++, so it only runs when TARPOLYGLOT_EXAMPLES=true
# says a compiler reachable by R is available. tar_dir() runs the code in a
# temporary directory.
if (identical(Sys.getenv("TARPOLYGLOT_EXAMPLES"), "true")) {
  targets::tar_dir({
    # Normally invoked by tar_target_cpp(); shown here as a direct call.
    writeLines(
      c("#include <Rcpp.h>",
        "// [[Rcpp::export]]",
        "double square(double x) { return x * x; }"),
      "square.cpp"
    )
    writeLines("square(x)", "post.R")
    run_cpp_step(
      script = "square.cpp",
      inputs = list(x = 21),
      post_script = "post.R"
    )
  })
}
```

run_jl_step

Execute a Julia step (worker behind tar_target_jl)

Description

Julia counterpart of `run_py_step()`. Runs, inside a fresh R environment, an optional R pre-script, a Python script (via **JuliaCall**). This is the function the target built by `tar_target_jl()` calls; it is exported so that call resolves at pipeline run time but this function is not destined to be called directly by the package users.

Usage

```
run_jl_step(
  script,
  pre_script = NULL,
  post_script = NULL,
  inputs = list(),
```

```

    output = "object",
    retrieve = NULL,
    files = NULL,
    julia_version = NULL,
    julia_home = getOption("tarpolyglot.julia_home"),
    julia_project = NULL,
    julia_packages = NULL,
    name = NULL
  )

```

Arguments

script	Path to the Julia script to run (required). Accepts a file path or an inline <code>tar_code()</code> carrier; see the "Script arguments" section below.
pre_script	Optional path to an R script run before the Julia script. It is evaluated in the step environment, which already holds the named inputs. To hand objects to Julia, assign a named list to <code>_jl</code> in this script; each element is <code>julia_assign()</code> ed as a variable in Julia's Main module. Accepts a file path or an inline <code>tar_code()</code> carrier; see the "Script arguments" section below.
post_script	Optional path to an R script run after the Julia script. It is evaluated in the same environment, which now also holds <code>jl_get(name)</code> and <code>jl_call(fn, ...)</code> . In <code>output = "object"</code> mode the value of its last expression becomes the target value; in <code>output = "file"</code> mode it must return a character vector of file paths. Accepts a file path or an inline <code>tar_code()</code> carrier; see the "Script arguments" section below.
inputs	Named list of upstream target values, bound by name into the step environment. Supplied automatically by the constructor.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").
retrieve	Optional character vector of foreign-session variable names to return when no <code>post_script</code> is supplied in object mode. One name returns that object; several return a named list.
files	Optional character vector of file paths to return when no <code>post_script</code> is supplied in file mode.
julia_version	Optional Julia version to select (e.g. "1.11"), used when <code>julia_home</code> is not given. Resolved to a juliaup -managed install. Default NULL uses the computer/global default Julia.
julia_home, julia_project, julia_packages	Julia environment selection. <code>julia_home</code> is the directory containing the Julia executable (defaults to <code>getOption("tarpolyglot.julia_home")</code> ; when unset and no <code>julia_version</code> , JuliaCall discovers Julia on PATH). <code>julia_project</code> is a Julia project environment (folder with <code>Project.toml</code> / <code>Manifest.toml</code>) to <code>Pkg.activate()</code> ; when NULL, Julia's default global environment (<code>@v#.#</code>) is used. <code>julia_packages</code> is a character vector of packages to using before the script. The requested environment takes priority over an ambient <code>JULIA_PROJECT</code> environment variable (e.g. one set by an RStudio project config and inherited

by crew workers): it is cleared for the duration of the Julia binding, so an explicit `julia_project` (or the global environment you get when none is given) wins over it, rather than `JULIA_PROJECT` silently selecting a different project. (`JULIA_HOME` is not cleared: it is a supported way to point at the default Julia.)

name Character string, the step's target name. Supplied automatically by the constructor; used only to name this step's log files when `polyglot_controller()` was given a `tar_polyglot_log()` (NULL – the default – disables logging for a direct call).

Value

The converted R object (object mode) or a character vector of normalised file paths (file mode).

Script arguments

A worker receives whatever the constructor already resolved, which is one of two things: a **path to a file** on disk, or an **inline carrier** built by `tar_code()` that holds the code in memory. Both are accepted, so a direct call may pass either.

`tar_target_path()` is deliberately *not* a third form at this level. It is a constructor-level convenience: `tar_target_py()` and the other constructors rewrite it while the pipeline's DAG is built, so that by the time a worker runs it has already become the ordinary file path held by the upstream target. Handing the result of `tar_target_path()` straight to a worker therefore does not resolve to a file. The three forms as written in `_targets.R`, and which of them tracks your edits, are covered in `vignette("scripts")`.

See Also

`run_py_step()`, `tar_target_jl()`

Examples

```
# This worker starts a live Julia session, so it only runs when
# TARPOLYGLOT_EXAMPLES=true says a Julia toolchain is available.
# tar_dir() runs the code in a temporary directory.
if (identical(Sys.getenv("TARPOLYGLOT_EXAMPLES"), "true")) {
  targets::tar_dir({
    # Normally invoked by tar_target_jl(); shown here as a direct call.
    writeLines("to_jl <- list(x = x)", "pre.R")
    writeLines("result = sum(x)", "sum.jl")
    run_jl_step(
      script = "sum.jl",
      pre_script = "pre.R",
      inputs = list(x = c(1, 2, 3)),
      retrieve = "result"
    )
  })
}
```

run_py_step

*Execute a Python step (worker behind tar_target_py)***Description**

Runs, inside a fresh R environment, an optional R pre-script, a Python script (via **reticulate**), and an optional R post-script, then returns either a converted R object or a character vector of files. This is the function the target built by `tar_target_py()` calls; it is exported so that call resolves at pipeline run time but this function is not destined to be called directly by the package users.

Usage

```
run_py_step(
  script,
  pre_script = NULL,
  post_script = NULL,
  inputs = list(),
  output = "object",
  retrieve = NULL,
  files = NULL,
  python_version = NULL,
  env = NULL,
  env_manager = "system",
  python = NULL,
  name = NULL
)
```

Arguments

script	Path to the Python script to run (required). Accepts a file path or an inline <code>tar_code()</code> carrier; see the "Script arguments" section below.
pre_script	Optional path to an R script run before the Python script. It is evaluated in the step environment, which already holds the named inputs. To hand objects to Python, assign a named list to <code>_py</code> in this script; each element is pushed as a top-level variable in the Python <code>__main__</code> module. Accepts a file path or an inline <code>tar_code()</code> carrier; see the "Script arguments" section below.
post_script	Optional path to an R script run after the Python script. It is evaluated in the same environment, which now also holds <code>py</code> (the <code>reticulate</code> <code>__main__</code> module proxy) and <code>py_get(name)</code> . In <code>output = "object"</code> mode the value of its last expression becomes the target value; in <code>output = "file"</code> mode it must return a character vector of file paths. Accepts a file path or an inline <code>tar_code()</code> carrier; see the "Script arguments" section below.
inputs	Named list of upstream target values, bound by name into the step environment. Supplied automatically by the constructor.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").

retrieve	Optional character vector of foreign-session variable names to return when no <code>post_script</code> is supplied in object mode. One name returns that object; several return a named list.
files	Optional character vector of file paths to return when no <code>post_script</code> is supplied in file mode.
python_version	Optional Python version to select (e.g. "3.12" or ">=3.11"), used only when no environment and no explicit python path are given. <code>reticulate</code> fetches/selects it (via its uv-backed ephemeral environment) with <code>reticulate::py_require()</code> . Default NULL uses the computer/global default Python. Use this when you only care about the interpreter <i>version</i> and are happy for <code>reticulate</code> to build a <i>throwaway</i> environment; for a <i>pinned, reproducible</i> environment use <code>env / env_manager</code> (or <code>python</code>) instead.
env, env_manager, python	Python environment selection: the reproducible alternatives to <code>python_version</code> . Use <code>env + env_manager</code> to point at an existing environment built by a known tool, or <code>python</code> for one explicit interpreter path. <code>env_manager</code> is one of "system", "virtualenv", "venv", "conda", "uv", "poetry"; <code>env</code> is the corresponding virtualenv/conda name or path (or poetry project directory); <code>python</code> is an explicit interpreter path. Precedence: <code>python > environment (env/env_manager) > python_version > default</code> . ("virtualenv", "venv" and "uv" all point at a standard virtualenv, including one created by <code>renv::use_python()</code> , and behave identically.) For "virtualenv"/"venv"/"uv"/"poetry", an <code>env</code> that is an already-existing directory (relative to the working directory, or absolute) is resolved to an absolute path before use, so a relative venv path (e.g. ".venv", created with <code>uv venv .venv</code>) works as expected; otherwise <code>reticulate::use_virtualenv()</code> would misread a separator-less relative path as the <i>name</i> of an environment under its own virtualenv root instead of a path on disk. A bare name that does not correspond to an existing directory is passed through unchanged, so a genuine named environment (one already registered under that root) still resolves. When <code>python</code> or an environment is given, the selection also takes priority over an ambient <code>RETICULATE_PYTHON</code> environment variable (e.g. one set by the RStudio project Python config and inherited by crew workers), which <code>reticulate</code> would otherwise let silently override the request.
name	Character string, the step's target name. Supplied automatically by the constructor; used only to name this step's log files when <code>polyglot_controller()</code> was given a <code>tar_polyglot_log()</code> (NULL – the default – disables logging for a direct call).

Value

The converted R object (object mode) or a character vector of normalised file paths (file mode).

Script arguments

A worker receives whatever the constructor already resolved, which is one of two things: a **path to a file** on disk, or an **inline carrier** built by `tar_code()` that holds the code in memory. Both are accepted, so a direct call may pass either.

`tar_target_path()` is deliberately *not* a third form at this level. It is a constructor-level convenience: `tar_target_py()` and the other constructors rewrite it while the pipeline's DAG is built, so that by the time a worker runs it has already become the ordinary file path held by the upstream target. Handing the result of `tar_target_path()` straight to a worker therefore does not resolve to a file. The three forms as written in `_targets.R`, and which of them tracks your edits, are covered in vignette("scripts").

See Also

`run_jl_step()`, `tar_target_py()`

Examples

```
# This worker binds a live Python interpreter, so it only runs when
# TARPOLYGLOT_EXAMPLES=true says a Python toolchain is available.
# tar_dir() runs the code in a temporary directory, so nothing is written
# to the working directory.
if (identical(Sys.getenv("TARPOLYGLOT_EXAMPLES"), "true")) {
  targets::tar_dir({
    # Normally invoked by tar_target_py(); shown here as a direct call.
    writeLines("to_py <- list(x = x)", "pre.R")
    writeLines("result = float(sum(x))", "sum.py")
    run_py_step(
      script = "sum.py",
      pre_script = "pre.R",
      inputs = list(x = c(1, 2, 3)),
      retrieve = "result"
    )
  })
}
```

run_rs_step

Execute a Rust step (worker behind tar_target_rs)

Description

Compiles the `#[extendr]` functions in a Rust script with `rextendr::rust_source()`, exposing them as R functions in a fresh environment, then evaluates an R **post-script** in that environment where you call those functions and return the result. Upstream inputs are bound in the same environment. This is the function the target built by `tar_target_rs()` calls; it is exported so the call resolves at run time, but package users should not call it directly.

Usage

```
run_rs_step(
  script,
  post_script = NULL,
  inputs = list(),
  output = "object",
```

```

    files = NULL,
    dependencies = NULL,
    features = NULL,
    profile = NULL,
    toolchain = NULL
  )

```

Arguments

script	Path to the Rust script containing <code>#[extendr]</code> functions (required). Accepts a file path or an inline <code>tar_code()</code> carrier; see the "Script arguments" section below.
post_script	Path to an R script evaluated after compilation. The compiled Rust functions and the named inputs are in scope; its last expression is the target value (object mode), or it returns a character vector of file paths (file mode). Required for object mode. Accepts a file path or an inline <code>tar_code()</code> carrier; see the "Script arguments" section below.
inputs	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").
files	Optional character vector of file paths to return when no <code>post_script</code> is supplied in file mode.
dependencies, features, profile	Passed to <code>extendr::rust_source()</code> : crate dependencies (named list), Cargo features, and build profile (e.g. "dev" or "release").
toolchain	Optional rustup toolchain (e.g. "stable-x86_64-pc-windows-gnu"); sets <code>RUSTUP_TOOLCHAIN</code> for the build. Default NULL uses the rustup default toolchain.

Details

Unlike Python/Julia there is **no pre-script** for Rust and no live interpreter: `rust_source()` compiles a dynamic library and R calls into it with real type conversion (via `extendr`). A Rust toolchain and cargo must be reachable (this function puts R, cargo, and on Windows Rtools on PATH for the build itself); on Windows use the GNU toolchain.

Value

The value of the post-script (object mode) or a character vector of normalised file paths (file mode).

Script arguments

A worker receives whatever the constructor already resolved, which is one of two things: a **path to a file** on disk, or an **inline carrier** built by `tar_code()` that holds the code in memory. Both are accepted, so a direct call may pass either.

`tar_target_path()` is deliberately *not* a third form at this level. It is a constructor-level convenience: `tar_target_py()` and the other constructors rewrite it while the pipeline's DAG is built, so that by the time a worker runs it has already become the ordinary file path held by the upstream target. Handing the result of `tar_target_path()` straight to a worker therefore does not resolve to a file. The three forms as written in `_targets.R`, and which of them tracks your edits, are covered in `vignette("scripts")`.

See Also

`tar_target_rs()`, `run_py_step()`, `run_jl_step()`

Examples

```
# This worker compiles Rust with cargo, so it only runs when
# TARPOLYGLOT_EXAMPLES=true says a Rust toolchain is available.
# tar_dir() runs the code in a temporary directory.
if (identical(Sys.getenv("TARPOLYGLOT_EXAMPLES"), "true")) {
  targets::tar_dir({
    # Normally invoked by tar_target_rs(); shown here as a direct call.
    writeLines("#[extendr] fn square(x: f64) -> f64 { x * x }", "square.rs")
    writeLines("square(x)", "post.R")
    run_rs_step(
      script = "square.rs",
      inputs = list(x = 21),
      post_script = "post.R"
    )
  })
}
```

tar_code

Inline code for a tarpolyglot step

Description

Use in place of a file-path string (or a `tar_target_path()` reference) for the `script`, `pre_script`, or `post_script` argument of the `tar_target_py()` / `tar_target_jl()` / `tar_target_rs()` constructors (and their `_raw` forms), to supply the code inline instead of pointing at a file.

Usage

```
tar_code(x)
```

Arguments

`x` Inline code: an R `{...}` block, or an expression/variable/literal that yields a single character string of source code.

Details

`tar_code()` accepts two forms. An R `{...}` block is captured as inline R code and is only valid in an R slot (`pre_script` or `post_script`); passing a block as the foreign script is an error. Anything that evaluates to a single character string is treated as inline source code, and its language follows from the slot it fills: the foreign script slot is Python, Julia, or Rust source, while a `pre_script` or `post_script` slot is R source.

Multi-line strings are supported and preserved verbatim, then **dedented**: the common leading-whitespace margin shared by all lines is stripped (and leading/ trailing blank lines dropped), so code you indent to line up with the surrounding `_targets.R` still starts flush-left. This matters for Python, whose top-level indentation is significant; Julia/Rust/R are unaffected either way. For Python or regex payloads, prefer an R raw string `r"(...)"` so backslashes and quotes need no escaping.

Because the inline code is embedded in the target's command, targets hashes it: editing inline code re-runs the target automatically (unlike a literal path string, which is untracked).

Value

A classed marker (`tp_inline`) recognised by the `tar_target_*` constructors.

See Also

[tar_target_path\(\)](#), [tar_target_py\(\)](#), [tar_target_jl\(\)](#), [tar_target_rs\(\)](#)

Examples

```
# Building a target does not run it, so these examples need no Python.
# Inline R (pre/post) plus a one-line inline Python `script`:
tarpolyglot::tar_target_py(
  name      = m,
  inputs    = c(x = "prepared_x"),
  pre_script = tar_code({ to_py <- list(x = x) }), # inline R
  script     = tar_code("result = float(sum(x))"), # inline Python
  post_script = tar_code({ py_get("result") })
)

# Multi-line inline Python. Indent the code to line up with `_targets.R`;
# tar_code() dedents it, so Python's own block indentation stays correct.
tarpolyglot::tar_target_py(
  name      = pos_sum,
  inputs    = c(x = "prepared_x"),
  script     = tar_code(r"(
    result = 0
    for v in x:
      if v > 0:
        result += v
  )")
)
```

tar_polyglot_log	<i>Configure per-step stdout/stderr logging</i>
------------------	---

Description

Passed to `polyglot_controller()`'s `log` argument to write one log file per Python/Julia step: `<name>.out` for stdout, `<name>.err` for stderr, named after the target. **crew** launches worker processes before it knows which target they will run, so this configuration cannot be applied at worker-launch time; instead `polyglot_controller()` stashes it as environment variables (inherited by every worker process it spawns, the same way any other environment variable is), and `run_py_step()` / `run_jl_step()` read them back and do the actual redirection once they know which step is running and what interpreter it resolved.

Usage

```
tar_polyglot_log(
  stdout = "./logs/out",
  stderr = "./logs/err",
  append = FALSE,
  header = TRUE
)
```

Arguments

<code>stdout, stderr</code>	Directory to write per-step log files into (created if missing), or NULL to disable that stream. Default <code>"./logs/out"</code> / <code>"./logs/err"</code> .
<code>append</code>	If FALSE (default), a step's log file is truncated at the start of that step's run, so it only ever holds the latest run's output. If TRUE, new output is appended after two blank lines separating it from any prior content, so the file accumulates history across repeated runs.
<code>header</code>	If TRUE (default), the stdout file gets a header written before the step's own output: the step name, <code>date()</code> , the resolved interpreter's version and path, and whether an explicit environment was used.

Details

Rust steps (`tar_target_rs()`) are not covered: **rextendr**-compiled code writes straight to the OS file descriptor, bypassing the redirection mechanism `reticulate/JuliaCall` provide for their embedded interpreters. Use **crew**'s own `options_local(log_directory = ...)` (process-level logging; since `polyglot_controller()` defaults to `tasks_max = 1`, each worker runs exactly one step, so that already gives one log per step, Rust included) if you need Rust step output.

C++ steps (`tar_target_cpp()`) **are** covered, with one caveat: `Rcpp::Rcout/Rcpp::Rcerr` (the idiomatic Rcpp printing calls) route through R's own output-connection system, so they are captured by an R-level `sink()` redirect the same way `cat()/message()` output is (confirmed empirically, including from a fresh **crew** worker process). Raw `std::cout/std::cerr/printf()` writes by-pass R's connection system entirely and write straight to the OS file descriptor, exactly like Rust's

`println!()` – **not** captured here; use `Rcpp::Rcout/Rcpp::Rcerr` in compiled code instead of raw C++ streams if you want step output in these logs.

Branches of one pattern share a single log file, language-agnostically. The log file name is fixed to the *target*’s name at the moment its command is built, before branching happens – so every branch of a `map()/tarpolyglot_map()`-driven step (Python, Julia, or C++ alike) writes to the same `<name>.out/<name>.err`, and since `append = FALSE` truncates at the start of *each* branch’s run, only the last branch to run leaves its output in the file; earlier branches’ output is overwritten, not lost-and-gone from disk but never actually visible. Not specific to C++ – confirmed while investigating C++ logging, but the same file-per-target-name design applies to every constructor. Use `append = TRUE` to at least keep all branches’ output (separated, in run order) instead of losing all but the last, or crew’s own options `local(log_directory = ...)` for a genuinely one-file-per-worker-process log.

Value

A list with class `tp_log`, for `polyglot_controller(log = ...)`.

See Also

[polyglot_controller\(\)](#)

Examples

```
# tar_polyglot_log() only builds a configuration object; nothing is written
# and no directory is created until a step actually runs.
log <- tar_polyglot_log(stdout = "./logs/out", stderr = "./logs/err")
str(log)

# It is passed to polyglot_controller(), which every Python/Julia/C++ step
# then inherits from.
controller <- polyglot_controller(log = log)
inherits(controller, "crew_class_controller")
```

tar_target_cpp

Target that runs a C++ script

Description

Non-standard-evaluation constructor mirroring [targets::tar_target\(\)](#) for C++: pass a bare name and unquoted pattern, for direct use in `_targets.R`. Compiles the `// [[Rcpp::export]]` functions in script with `Rcpp::sourceCpp()` and calls them from the R `post_script`. Delegates to [tar_target_cpp_raw\(\)](#); see it and [run_cpp_step\(\)](#) for the full reference. The Rust twin is [tar_target_rs\(\)](#); Python/Julia are [tar_target_py\(\)](#) / [tar_target_jl\(\)](#).

Usage

```

tar_target_cpp(
  name,
  script,
  post_script = NULL,
  inputs = NULL,
  output = "object",
  files = NULL,
  depends = NULL,
  pattern = NULL,
  packages = targets::tar_option_get("packages"),
  library = targets::tar_option_get("library"),
  deps = NULL,
  string = NULL,
  format = NULL,
  repository = targets::tar_option_get("repository"),
  iteration = targets::tar_option_get("iteration"),
  error = targets::tar_option_get("error"),
  memory = targets::tar_option_get("memory"),
  garbage_collection = isTRUE(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"),
  description = targets::tar_option_get("description")
)

```

Arguments

name	Symbol, the target name (unquoted).
script	Path to the C++ script with <code>// [[Rcpp::export]]</code> functions (required). Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
post_script	Path to an R script run after compilation, where the compiled functions and inputs are in scope. Its last expression is the value (object mode); it returns file paths (file mode). Required for object mode. Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
inputs	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").

files	Optional character vector of file paths to return when no <code>post_script</code> is supplied in file mode.
depends	Optional character vector of extension packages (e.g. <code>c("RcppArmadillo", "RcppEigen")</code>); see <code>run_cpp_step()</code> .
pattern	Optional branching pattern as described in the targets package documentation , unquoted (e.g. <code>map(x)</code>). The patterns included in the targets package (<code>map()</code> , <code>head()</code> , ...) are accepted, but it is recommended to use the tarpolyglot pattern functions instead, as they compile the library once and reuse it across the branches (see <code>tarpolyglot_map()</code> , <code>tarpolyglot_head()</code> , <code>tarpolyglot_tail()</code> , <code>tarpolyglot_cross()</code> , <code>tarpolyglot_slice()</code> , <code>tarpolyglot_sample()</code>).
packages, library	Character vectors of R packages (and library paths) to load for the target, forwarded to <code>targets::tar_target_raw()</code> .
deps, string	Advanced <code>targets::tar_target_raw()</code> arguments: extra dependency names and a string used for change detection.
format, repository, iteration	Storage/iteration settings forwarded to <code>targets::tar_target_raw()</code> . <code>format</code> defaults to "file" when <code>output = "file"</code> , otherwise to the <code>targets</code> option default.
error, memory, garbage_collection, deployment, priority, resources, storage, retrieval, cue, description	Standard <code>targets::tar_target_raw()</code> execution/behaviour arguments, forwarded unchanged.

Value

A `targets` target object. When `pattern` uses `tarpolyglot_map()` it is instead a list of two targets: the `<name>_cpp_lib` compile target and the branched `<name>` target.

Script options

Every script argument (`script`, and `pre_script` / `post_script` where the constructor has them) accepts the same three forms. The choice is not cosmetic: it decides whether editing the code re-runs the target.

A literal path string e.g. `script = "py/step.py"`. The file is read when the step runs, but it is **not** tracked, so editing it does **not** invalidate the target: `targets` will happily reuse a stale result until some *other* dependency changes. Simplest form, and a reasonable default once a script has settled.

A `tar_target_path()` reference e.g. `script = tar_target_path("step_py")`, naming an upstream `tar_target(step_py, "py/step.py", format = "file")`. The file becomes a real `targets` dependency, so editing it **does** invalidate this target and the step re-runs. This is what you want while a script is still changing, and the recommended form for reproducible pipelines.

Inline code via `tar_code()` e.g. `script = tar_code("result = sum(x)")`. The code lives in `_targets.R` rather than in a file, and is embedded in the target's command, so `targets` hashes it and editing it **does** invalidate the target. A character string carries foreign source (Python,

Julia, Rust, C++) or R source; an R { ... } block carries inline R and is accepted only by pre_script / post_script, never by the foreign script.

Mixing forms in one call is fine: a tracked script with a literal post_script, inline code for one and a file for another, and so on. See vignette("scripts") for worked examples of all three and guidance on choosing.

See Also

[tar_target_cpp_raw\(\)](#), [tarpolyglot_map\(\)](#), [run_cpp_step\(\)](#), [tar_target_rs\(\)](#), [tar_target_py\(\)](#), [tar_target_jl\(\)](#)

Examples

```
# Building a target does not run it, so these examples need no C++ compiler.
# scripts/square.cpp:
#   // [[Rcpp::export]]
#   double square(double x) { return x * x; }
# scripts/post.R:
#   square(x)
list(
  tarpolyglot::tar_target_cpp(
    name = cpp_square,
    script = "scripts/square.cpp",
    inputs = c(x = "value"),
    post_script = "scripts/post.R"
  )
)

# The three ways to supply a script (see the "Script options" section):
# 1. Literal path: untracked, editing the file does NOT re-run the step.
tarpolyglot::tar_target_cpp(
  name = demo_literal, script = "cpp/step.cpp", post_script = "R/post.R"
)

# 2. tar_target_path(): tracked, editing the file DOES re-run the step.
list(
  targets::tar_target(step_cpp, "cpp/step.cpp", format = "file"),
  tarpolyglot::tar_target_cpp(
    name = demo_tracked,
    script = tarpolyglot::tar_target_path("step_cpp"),
    post_script = "R/post.R"
  )
)

# 3. tar_code(): inline, editing the code DOES re-run the step. A string
#   carries foreign source; an R { } block carries inline R.
tarpolyglot::tar_target_cpp(
  name = demo_inline,
  script = tarpolyglot::tar_code(
    "/* [[Rcpp::export]]\nint one() { return 1; }"
  ),
)
```

```

    post_script = tarpolyglot::tar_code({ one() })
  )

```

tar_target_cpp_raw	<i>Target that runs a C++ script (raw / factory form)</i>
--------------------	---

Description

Character-based constructor mirroring `targets::tar_target_raw()` for C++, for use inside targets factories. Returns a single targets target whose command compiles the `// [[Rcpp::export]]` functions in script with `Rcpp::sourceCpp()` and then evaluates an R **post-script** that calls those functions (with the upstream inputs in scope) and returns the value. See `run_cpp_step()` for the contract. For direct use in `_targets.R`, see `tar_target_cpp()`.

Usage

```

tar_target_cpp_raw(
  name,
  script,
  post_script = NULL,
  inputs = NULL,
  output = "object",
  files = NULL,
  depends = NULL,
  pattern = NULL,
  packages = targets::tar_option_get("packages"),
  library = targets::tar_option_get("library"),
  deps = NULL,
  string = NULL,
  format = NULL,
  repository = targets::tar_option_get("repository"),
  iteration = targets::tar_option_get("iteration"),
  error = targets::tar_option_get("error"),
  memory = targets::tar_option_get("memory"),
  garbage_collection = isTRUE(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"),
  description = targets::tar_option_get("description")
)

```

Arguments

name	Character string, the target name.
------	------------------------------------

script	Path to the C++ script with // <code>[[Rcpp::export]]</code> functions (required). Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
post_script	Path to an R script run after compilation, where the compiled functions and inputs are in scope. Its last expression is the value (object mode); it returns file paths (file mode). Required for object mode. Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
inputs	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").
files	Optional character vector of file paths to return when no <code>post_script</code> is supplied in file mode.
depends	Optional character vector of extension packages (e.g. <code>c("RcppArmadillo", "RcppEigen")</code>); see <code>run_cpp_step()</code> .
pattern	Optional branching pattern as described in the targets package documentation , as a language object (e.g. <code>quote(map(x))</code>), forwarded to <code>targets::tar_target_raw()</code> . The patterns included in the targets package (<code>map()</code> , <code>head()</code> , ...) are accepted, but it is recommended to use the tarpolyglot pattern functions instead, as they compile the library once and reuse it across the branches (see <code>tarpolyglot_map()</code> , <code>tarpolyglot_head()</code> , <code>tarpolyglot_tail()</code> , <code>tarpolyglot_cross()</code> , <code>tarpolyglot_slice()</code> , <code>tarpolyglot_sample()</code>).
packages, library	Character vectors of R packages (and library paths) to load for the target, forwarded to <code>targets::tar_target_raw()</code> .
deps, string	Advanced <code>targets::tar_target_raw()</code> arguments: extra dependency names and a string used for change detection.
format, repository, iteration	Storage/iteration settings forwarded to <code>targets::tar_target_raw()</code> . <code>format</code> defaults to "file" when <code>output = "file"</code> , otherwise to the <code>targets</code> option default.
error, memory, garbage_collection, deployment, priority, resources, storage, retrieval, cue, description	Standard <code>targets::tar_target_raw()</code> execution/behaviour arguments, forwarded unchanged.

Details

Unlike Python/Julia there is **no pre-script** for C++: inputs are used directly in the post-script alongside the compiled functions, mirroring `tar_target_rs()`. A working C++ compiler is required; `sourceCpp()` compiles via R's own configured toolchain (R CMD SHLIB), so, unlike Rust, there is no separate compiler/ABI to match – on Windows Rtools must be installed and discoverable the same way it already needs to be for any R package with compiled code.

Under dynamic branching, `pattern = map(...)` recompiles the library in every branch (C++ has no live interpreter to reuse, the same limitation Rust has). Passing a `tarpolyglot` pattern helper instead (`tarpolyglot_map()`, `tarpolyglot_cross()`, `tarpolyglot_slice()`, `tarpolyglot_head()`, `tarpolyglot_tail()`, `tarpolyglot_sample()`) compiles the library **once** in a companion target named `<name>_cpp_lib` and reuses it across all branches; the constructor then returns *both* targets as a list. See `tarpolyglot_map()`.

Value

A targets target object. When `pattern` uses `tarpolyglot_map()` it is instead a list of two targets: the `<name>_cpp_lib` compile target and the branched `<name>` target.

Script options

Every script argument (`script`, and `pre_script` / `post_script` where the constructor has them) accepts the same three forms. The choice is not cosmetic: it decides whether editing the code re-runs the target.

A literal path string e.g. `script = "py/step.py"`. The file is read when the step runs, but it is **not** tracked, so editing it does **not** invalidate the target: targets will happily reuse a stale result until some *other* dependency changes. Simplest form, and a reasonable default once a script has settled.

A `tar_target_path()` reference e.g. `script = tar_target_path("step_py")`, naming an upstream `tar_target(step_py, "py/step.py", format = "file")`. The file becomes a real targets dependency, so editing it **does** invalidate this target and the step re-runs. This is what you want while a script is still changing, and the recommended form for reproducible pipelines.

Inline code via `tar_code()` e.g. `script = tar_code("result = sum(x)")`. The code lives in `_targets.R` rather than in a file, and is embedded in the target's command, so targets hashes it and editing it **does** invalidate the target. A character string carries foreign source (Python, Julia, Rust, C++) or R source; an R `{ ... }` block carries inline R and is accepted only by `pre_script` / `post_script`, never by the foreign script.

Mixing forms in one call is fine: a tracked script with a literal `post_script`, inline code for one and a file for another, and so on. See `vignette("scripts")` for worked examples of all three and guidance on choosing.

See Also

`tar_target_cpp()`, `tarpolyglot_map()`, `run_cpp_step()`, `tar_target_rs_raw()`, `tar_target_py_raw()`, `tar_target_jl_raw()`

Examples

```
# Building a target does not run it, so these examples need no C++ compiler.
# scripts/square.cpp:
#   // [[Rcpp::export]]
#   double square(double x) { return x * x; }
# scripts/post.R:
#   square(x)
tarpolyglot::tar_target_cpp_raw(
```

```

    name = "cpp_square",
    script = "scripts/square.cpp",
    inputs = c(x = "value"),
    post_script = "scripts/post.R"
  )

# The three ways to supply a script (see the "Script options" section):
# 1. Literal path: untracked, editing the file does NOT re-run the step.
tarpolyglot::tar_target_cpp_raw(
  name = "demo_literal", script = "cpp/step.cpp", post_script = "R/post.R"
)

# 2. tar_target_path(): tracked, editing the file DOES re-run the step.
list(
  targets::tar_target(step_cpp, "cpp/step.cpp", format = "file"),
  tarpolyglot::tar_target_cpp_raw(
    name = "demo_tracked",
    script = tarpolyglot::tar_target_path("step_cpp"),
    post_script = "R/post.R"
  )
)

# 3. tar_code(): inline, editing the code DOES re-run the step. A string
#   carries foreign source; an R { } block carries inline R.
tarpolyglot::tar_target_cpp_raw(
  name = "demo_inline",
  script = tarpolyglot::tar_code(
    "// [[Rcpp::export]]\nint one() { return 1; }"
  ),
  post_script = tarpolyglot::tar_code({ one() })
)

```

tar_target_jl

Target that runs a Julia script

Description

Non-standard-evaluation constructor mirroring `targets::tar_target()`: pass a bare name and unquoted pattern, for direct use in `_targets.R`. Delegates to `tar_target_jl_raw()`; see it and `run_jl_step()` for the full reference. The Python twin is `tar_target_py()`.

Usage

```

tar_target_jl(
  name,
  script,
  pre_script = NULL,
  post_script = NULL,
  inputs = NULL,

```

```

output = "object",
retrieve = NULL,
files = NULL,
julia_version = NULL,
julia_home = getOption("tarpolyglot.julia_home"),
julia_project = NULL,
julia_packages = NULL,
pattern = NULL,
packages = targets::tar_option_get("packages"),
library = targets::tar_option_get("library"),
deps = NULL,
string = NULL,
format = NULL,
repository = targets::tar_option_get("repository"),
iteration = targets::tar_option_get("iteration"),
error = targets::tar_option_get("error"),
memory = targets::tar_option_get("memory"),
garbage_collection = isTRUE(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"),
description = targets::tar_option_get("description")
)

```

Arguments

name	Symbol, the target name (unquoted).
script	Path to the Julia script to run (required). Accepts a literal path, a tar_target_path() reference, or inline code from tar_code() ; see the "Script options" section below.
pre_script	Optional path to an R script run before the Julia script. See run_jl_step() ; assign a named list to <code>_jl</code> to hand objects to Julia. Accepts a literal path, a tar_target_path() reference, or inline code from tar_code() ; see the "Script options" section below.
post_script	Optional path to an R script run after the Julia script. See run_jl_step() ; helpers <code>jl_get()</code> and <code>jl_call()</code> are available, and its last expression (object mode) or returned paths (file mode) become the value. Accepts a literal path, a tar_target_path() reference, or inline code from tar_code() ; see the "Script options" section below.
inputs	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.

output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").
retrieve	Optional character vector of foreign-session variable names to return when no post_script is supplied in object mode. One name returns that object; several return a named list.
files	Optional character vector of file paths to return when no post_script is supplied in file mode.
julia_version	Optional Julia version to select (e.g. "1.11"), used when julia_home is not given; resolved to a juliaup-managed install. Forwarded to <code>run_jl_step()</code> . Default NULL uses the computer/global Julia.
julia_home, julia_project, julia_packages	Julia environment selection, forwarded to <code>run_jl_step()</code> . <code>julia_home</code> is the directory containing the julia executable (defaults to <code>getOption("tarpolyglot.julia_home")</code> ; when unset and no <code>julia_version</code> , JuliaCall discovers Julia on PATH). <code>julia_project</code> is a Julia project environment to <code>Pkg.activate()</code> . <code>julia_packages</code> is a character vector of packages to using before the script.
pattern	Optional dynamic-branching pattern, unquoted (e.g. <code>map(x)</code>).
packages, library	Character vectors of R packages (and library paths) to load for the target, forwarded to <code>targets::tar_target_raw()</code> .
deps, string	Advanced <code>targets::tar_target_raw()</code> arguments: extra dependency names and a string used for change detection.
format, repository, iteration	Storage/iteration settings forwarded to <code>targets::tar_target_raw()</code> . <code>format</code> defaults to "file" when <code>output = "file"</code> , otherwise to the <code>targets</code> option default.
error, memory, garbage_collection, deployment, priority, resources, storage, retrieval, cue, description	Standard <code>targets::tar_target_raw()</code> execution/behaviour arguments, forwarded unchanged.

Value

A targets target object.

Script options

Every script argument (script, and pre_script / post_script where the constructor has them) accepts the same three forms. The choice is not cosmetic: it decides whether editing the code re-runs the target.

A literal path string e.g. `script = "py/step.py"`. The file is read when the step runs, but it is **not** tracked, so editing it does **not** invalidate the target: targets will happily reuse a stale result until some *other* dependency changes. Simplest form, and a reasonable default once a script has settled.

A `tar_target_path()` **reference** e.g. `script = tar_target_path("step_py")`, naming an upstream `tar_target(step_py, "py/step.py", format = "file")`. The file becomes a real targets dependency, so editing it **does** invalidate this target and the step re-runs. This is what you want while a script is still changing, and the recommended form for reproducible pipelines.

Inline code via `tar_code()` e.g. `script = tar_code("result = sum(x)")`. The code lives in `_targets.R` rather than in a file, and is embedded in the target's command, so targets hashes it and editing it **does** invalidate the target. A character string carries foreign source (Python, Julia, Rust, C++) or R source; an R `{ ... }` block carries inline R and is accepted only by `pre_script` / `post_script`, never by the foreign script.

Mixing forms in one call is fine: a tracked script with a literal `post_script`, inline code for one and a file for another, and so on. See `vignette("scripts")` for worked examples of all three and guidance on choosing.

See Also

`tar_target_jl_raw()`, `run_jl_step()`, `tar_target_py()`

Examples

```
# Building a target does not run it, so these examples need no Julia.
# scripts/sum.jl:
#   result = sum(x)
# scripts/post.R:
#   jl_get("result")
list(
  tarpolyglot::tar_target_jl(
    name = jl_sum,
    script = "scripts/sum.jl",
    inputs = c(x = "prepared_x"),
    post_script = "scripts/post.R"
  )
)

# The three ways to supply a script (see the "Script options" section):
# 1. Literal path: untracked, editing the file does NOT re-run the step.
tarpolyglot::tar_target_jl(
  name = demo_literal, script = "jl/step.jl", retrieve = "result"
)

# 2. tar_target_path(): tracked, editing the file DOES re-run the step.
list(
  targets::tar_target(step_jl, "jl/step.jl", format = "file"),
  tarpolyglot::tar_target_jl(
    name = demo_tracked,
    script = tarpolyglot::tar_target_path("step_jl"),
    retrieve = "result"
  )
)
```

```

# 3. tar_code(): inline, editing the code DOES re-run the step. A string
#   carries foreign source; an R { } block carries inline R.
tarpolyglot::tar_target_jl(
  name = demo_inline,
  script = tarpolyglot::tar_code("result = 1 + 1"),
  post_script = tarpolyglot::tar_code({ jl_get("result") })
)

# Tracking a helper file the script includes. Point `inputs` at a
# format = "file" target so editing the helper also re-runs the step;
# `inputs` takes the *target* name, not the path. Julia resolves a
# relative include() against the including file's own directory, so make
# the path absolute. jl/step.jl is then:
#   include(helper_path)
#   result = sum(myscale(x))
list(
  targets::tar_target(helper_file, "jl/helper.jl", format = "file"),
  tarpolyglot::tar_target_jl(
    name = demo_helper,
    script = "jl/step.jl",
    inputs = c(x = "prepared_x", helper_path = "helper_file"),
    pre_script = tarpolyglot::tar_code({
      to_jl <- list(x = x, helper_path = normalizePath(helper_path, winslash = "/"))
    }),
    retrieve = "result"
  )
)

```

tar_target_jl_raw

Target that runs a Julia script (raw / factory form)

Description

Character-based constructor mirroring `targets::tar_target_raw()`: name is a string and it is meant for use *inside targets factories*. Returns a single targets target whose command runs an optional R pre-script, script (via **JuliaCall**), and an optional R post-script, then returns a converted R object or a character vector of files. See `run_jl_step()` for the pre/post-script contract (the `to_jl` hand-off and the `jl_get` / `jl_call` helpers). For direct use in `_targets.R`, see `tar_target_jl()`. The Python twin is `tar_target_py_raw()`.

Usage

```

tar_target_jl_raw(
  name,
  script,
  pre_script = NULL,
  post_script = NULL,
  inputs = NULL,
  output = "object",

```

```

retrieve = NULL,
files = NULL,
julia_version = NULL,
julia_home = getOption("tarpolyglot.julia_home"),
julia_project = NULL,
julia_packages = NULL,
pattern = NULL,
packages = targets::tar_option_get("packages"),
library = targets::tar_option_get("library"),
deps = NULL,
string = NULL,
format = NULL,
repository = targets::tar_option_get("repository"),
iteration = targets::tar_option_get("iteration"),
error = targets::tar_option_get("error"),
memory = targets::tar_option_get("memory"),
garbage_collection = isTRUE(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"),
description = targets::tar_option_get("description")
)

```

Arguments

name	Character string, the target name.
script	Path to the Julia script to run (required). Accepts a literal path, a tar_target_path() reference, or inline code from tar_code() ; see the "Script options" section below.
pre_script	Optional path to an R script run before the Julia script. See run_jl_step() ; assign a named list to <code>to_jl</code> to hand objects to Julia. Accepts a literal path, a tar_target_path() reference, or inline code from tar_code() ; see the "Script options" section below.
post_script	Optional path to an R script run after the Julia script. See run_jl_step() ; helpers <code>jl_get()</code> and <code>jl_call()</code> are available, and its last expression (object mode) or returned paths (file mode) become the value. Accepts a literal path, a tar_target_path() reference, or inline code from tar_code() ; see the "Script options" section below.
inputs	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").

retrieve	Optional character vector of foreign-session variable names to return when no <code>post_script</code> is supplied in object mode. One name returns that object; several return a named list.
files	Optional character vector of file paths to return when no <code>post_script</code> is supplied in file mode.
julia_version	Optional Julia version to select (e.g. "1.11"), used when <code>julia_home</code> is not given; resolved to a juliaup-managed install. Forwarded to <code>run_jl_step()</code> . Default NULL uses the computer/global Julia.
julia_home, julia_project, julia_packages	Julia environment selection, forwarded to <code>run_jl_step()</code> . <code>julia_home</code> is the directory containing the julia executable (defaults to <code>getOption("tarpolyglot.julia_home")</code> ; when unset and no <code>julia_version</code> , JuliaCall discovers Julia on PATH). <code>julia_project</code> is a Julia project environment to <code>Pkg.activate()</code> . <code>julia_packages</code> is a character vector of packages to using before the script.
pattern	Optional targets dynamic-branching pattern as a language object (e.g. <code>quote(map(x))</code>), forwarded to <code>targets::tar_target_raw()</code> . The <code>tarpolyglot_map()</code> family is also accepted and behaves identically to the plain targets patterns here (they only differ on the Rust constructor).
packages, library	Character vectors of R packages (and library paths) to load for the target, forwarded to <code>targets::tar_target_raw()</code> .
deps, string	Advanced <code>targets::tar_target_raw()</code> arguments: extra dependency names and a string used for change detection.
format, repository, iteration	Storage/iteration settings forwarded to <code>targets::tar_target_raw()</code> . <code>format</code> defaults to "file" when <code>output = "file"</code> , otherwise to the targets option default.
error, memory, garbage_collection, deployment, priority, resources, storage, retrieval, cue, description	Standard <code>targets::tar_target_raw()</code> execution/behaviour arguments, forwarded unchanged.

Details

All `targets::tar_target_raw()` arguments are forwarded unchanged, so dynamic branching (pattern), storage format, deployment, resources, cue, etc. all work as usual. Upstream targets are wired in through inputs, which become dependencies (and are sliced under dynamic branching).

Value

A targets target object.

Script options

Every script argument (script, and `pre_script` / `post_script` where the constructor has them) accepts the same three forms. The choice is not cosmetic: it decides whether editing the code re-runs the target.

A literal path string e.g. `script = "py/step.py"`. The file is read when the step runs, but it is **not** tracked, so editing it does **not** invalidate the target: targets will happily reuse a stale result until some *other* dependency changes. Simplest form, and a reasonable default once a script has settled.

A `tar_target_path()` reference e.g. `script = tar_target_path("step_py")`, naming an upstream `tar_target(step_py, "py/step.py", format = "file")`. The file becomes a real targets dependency, so editing it **does** invalidate this target and the step re-runs. This is what you want while a script is still changing, and the recommended form for reproducible pipelines.

Inline code via `tar_code()` e.g. `script = tar_code("result = sum(x)")`. The code lives in `_targets.R` rather than in a file, and is embedded in the target's command, so targets hashes it and editing it **does** invalidate the target. A character string carries foreign source (Python, Julia, Rust, C++) or R source; an R `{ ... }` block carries inline R and is accepted only by `pre_script` / `post_script`, never by the foreign script.

Mixing forms in one call is fine: a tracked script with a literal `post_script`, inline code for one and a file for another, and so on. See `vignette("scripts")` for worked examples of all three and guidance on choosing.

See Also

`tar_target_jl()`, `run_jl_step()`, `tar_target_py_raw()`

Examples

```
# Building a target does not run it, so these examples need no Julia.
# scripts/sum.jl (uses the `using Statistics` julia_packages requests):
#   result = mean(x)
# scripts/post.R:
#   jl_get("result")
tarpolyglot::tar_target_jl_raw(
  name = "jl_sum",
  script = "scripts/sum.jl",
  inputs = c(x = "prepared_x"),
  post_script = "scripts/post.R",
  julia_packages = "Statistics"
)
```

```
# The three ways to supply a script (see the "Script options" section):
# 1. Literal path: untracked, editing the file does NOT re-run the step.
tarpolyglot::tar_target_jl_raw(
  name = "demo_literal", script = "jl/step.jl", retrieve = "result"
)
```

```
# 2. tar_target_path(): tracked, editing the file DOES re-run the step.
list(
  targets::tar_target(step_jl, "jl/step.jl", format = "file"),
  tarpolyglot::tar_target_jl_raw(
    name = "demo_tracked",
    script = tarpolyglot::tar_target_path("step_jl"),
    retrieve = "result"
  )
)
```

```

    )
  )

  # 3. tar_code(): inline, editing the code DOES re-run the step. A string
  #   carries foreign source; an R { } block carries inline R.
  tarpolyglot::tar_target_jl_raw(
    name = "demo_inline",
    script = tarpolyglot::tar_code("result = 1 + 1"),
    post_script = tarpolyglot::tar_code({ jl_get("result") })
  )

  # Tracking a helper file the script includes. Point `inputs` at a
  # format = "file" target so editing the helper also re-runs the step;
  # `inputs` takes the *target* name, not the path. Julia resolves a
  # relative include() against the including file's own directory, so make
  # the path absolute. jl/step.jl is then:
  #   include(helper_path)
  #   result = sum(myscale(x))
  list(
    targets::tar_target(helper_file, "jl/helper.jl", format = "file"),
    tarpolyglot::tar_target_jl_raw(
      name = "demo_helper",
      script = "jl/step.jl",
      inputs = c(x = "prepared_x", helper_path = "helper_file"),
      pre_script = tarpolyglot::tar_code({
        to_jl <- list(x = x, helper_path = normalizePath(helper_path, winslash = "/"))
      }),
      retrieve = "result"
    )
  )
)

```

tar_target_path	<i>Track a script argument as a targets dependency</i>
-----------------	--

Description

Use as the script, pre_script, or post_script argument of `tar_target_py()`/`tar_target_jl()`/`tar_target_rs()` (and their `_raw` forms) instead of a literal path string, to make that step re-run whenever the script file changes. name is the name of an upstream targets target (typically one created with format = "file" tracking the script file), and its *value* (the file path) is substituted in when the pipeline runs.

Usage

```
tar_target_path(name)
```

Arguments

name	Character string, the name of an upstream target whose value is the script's file path (e.g. a <code>tar_target(..., format = "file")</code>).
------	---

Details

This mirrors how `inputs = c(x = "some_target")` already wires an upstream target in by name: the target's name is given as a string, and the constructor turns it into a real dependency. A plain string passed directly as `script/pre_script/post_script` keeps meaning what it always has (an untracked literal path), so existing pipelines are unaffected.

Value

An object marking name for dependency-wiring by the `tar_target_*` constructors.

See Also

[tar_target_py\(\)](#), [tar_target_jl\(\)](#), [tar_target_rs\(\)](#)

Examples

```
# Building a target does not run it, so this example needs no Python.
# python/fit_step.py:
#   result = sum(x)
# scripts/pre.R:
#   to_py <- list(x = x)
list(
  targets::tar_target(fit_pyscript, "python/fit_step.py", format = "file"),
  tar_target_py(
    name = fit,
    script = tar_target_path("fit_pyscript"), # re-runs when the file changes
    pre_script = "scripts/pre.R",           # untracked literal path
    inputs = c(x = "prepared_x"),
    retrieve = "result"
  )
)
```

tar_target_py

Target that runs a Python script

Description

Non-standard-evaluation constructor mirroring `targets::tar_target()`: pass a bare name and an unquoted pattern, for direct use in `_targets.R`. It quotes those and delegates to `tar_target_py_raw()`. See that function and `run_py_step()` for the full argument reference and the pre/post-script contract. The Julia twin is `tar_target_jl()`.

Usage

```
tar_target_py(
  name,
  script,
  pre_script = NULL,
```

```

    post_script = NULL,
    inputs = NULL,
    output = "object",
    retrieve = NULL,
    files = NULL,
    python_version = NULL,
    env = NULL,
    env_manager = "system",
    python = NULL,
    pattern = NULL,
    packages = targets::tar_option_get("packages"),
    library = targets::tar_option_get("library"),
    deps = NULL,
    string = NULL,
    format = NULL,
    repository = targets::tar_option_get("repository"),
    iteration = targets::tar_option_get("iteration"),
    error = targets::tar_option_get("error"),
    memory = targets::tar_option_get("memory"),
    garbage_collection = isTRUE(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"),
    description = targets::tar_option_get("description")
  )

```

Arguments

name	Symbol, the target name (unquoted).
script	Path to the Python script to run (required). Accepts a literal path, a tar_target_path() reference, or inline code from tar_code() ; see the "Script options" section below.
pre_script	Optional path to an R script run before the Python script. See run_py_step() ; assign a named list to <code>to_py</code> to hand objects to Python. Accepts a literal path, a tar_target_path() reference, or inline code from tar_code() ; see the "Script options" section below.
post_script	Optional path to an R script run after the Python script. See run_py_step() ; helpers <code>py</code> and <code>py_get()</code> are available, and its last expression (object mode) or returned paths (file mode) become the value. Accepts a literal path, a tar_target_path() reference, or inline code from tar_code() ; see the "Script options" section below.
inputs	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a

	dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").
retrieve	Optional character vector of foreign-session variable names to return when no post_script is supplied in object mode. One name returns that object; several return a named list.
files	Optional character vector of file paths to return when no post_script is supplied in file mode.
python_version	Optional Python version to select (e.g. "3.12" or ">=3.11"), used only when no environment and no explicit python path are given. reticulate fetches/selects it (via its uv-backed ephemeral environment) with <code>reticulate::py_require()</code> . Default NULL uses the computer/global default Python. Use this when you only care about the interpreter <i>version</i> and are happy for reticulate to build a <i>throwaway</i> environment; for a <i>pinned, reproducible</i> environment use <code>env / env_manager</code> (or <code>python</code>) instead.
env, env_manager, python	Python environment selection: the reproducible alternatives to <code>python_version</code> . Use <code>env + env_manager</code> to point at an existing environment built by a known tool, or <code>python</code> for one explicit interpreter path. <code>env_manager</code> is one of "system", "virtualenv", "venv", "conda", "uv", "poetry"; <code>env</code> is the corresponding virtualenv/conda name or path (or poetry project directory); <code>python</code> is an explicit interpreter path. Precedence: <code>python > environment (env/env_manager) > python_version > default</code> . ("virtualenv", "venv" and "uv" all point at a standard virtualenv, including one created by <code>renv::use_python()</code> , and behave identically.) For "virtualenv"/"venv"/"uv"/"poetry", an <code>env</code> that is an already-existing directory (relative to the working directory, or absolute) is resolved to an absolute path before use, so a relative venv path (e.g. ".venv", created with <code>uv venv .venv</code>) works as expected; otherwise <code>reticulate::use_virtualenv()</code> would misread a separator-less relative path as the <i>name</i> of an environment under its own virtualenv root instead of a path on disk. A bare name that does not correspond to an existing directory is passed through unchanged, so a genuine named environment (one already registered under that root) still resolves. When <code>python</code> or an environment is given, the selection also takes priority over an ambient <code>RETICULATE_PYTHON</code> environment variable (e.g. one set by the RStudio project Python config and inherited by crew workers), which reticulate would otherwise let silently override the request.
pattern	Optional dynamic-branching pattern, unquoted (e.g. <code>map(x)</code>).
packages, library	Character vectors of R packages (and library paths) to load for the target, forwarded to <code>targets::tar_target_raw()</code> .
deps, string	Advanced <code>targets::tar_target_raw()</code> arguments: extra dependency names and a string used for change detection.
format, repository, iteration	Storage/iteration settings forwarded to <code>targets::tar_target_raw()</code> . <code>format</code> defaults to "file" when <code>output = "file"</code> , otherwise to the <code>targets</code> option default.

error, memory, garbage_collection, deployment, priority, resources,
storage, retrieval, cue, description

Standard `targets::tar_target_raw()` execution/behaviour arguments, forwarded unchanged.

Value

A targets target object.

Script options

Every script argument (script, and pre_script / post_script where the constructor has them) accepts the same three forms. The choice is not cosmetic: it decides whether editing the code re-runs the target.

A literal path string e.g. `script = "py/step.py"`. The file is read when the step runs, but it is **not** tracked, so editing it does **not** invalidate the target: targets will happily reuse a stale result until some *other* dependency changes. Simplest form, and a reasonable default once a script has settled.

A `tar_target_path()` reference e.g. `script = tar_target_path("step_py")`, naming an upstream `tar_target(step_py, "py/step.py", format = "file")`. The file becomes a real targets dependency, so editing it **does** invalidate this target and the step re-runs. This is what you want while a script is still changing, and the recommended form for reproducible pipelines.

Inline code via `tar_code()` e.g. `script = tar_code("result = sum(x)")`. The code lives in `_targets.R` rather than in a file, and is embedded in the target's command, so targets hashes it and editing it **does** invalidate the target. A character string carries foreign source (Python, Julia, Rust, C++) or R source; an R `{ ... }` block carries inline R and is accepted only by pre_script / post_script, never by the foreign script.

Mixing forms in one call is fine: a tracked script with a literal post_script, inline code for one and a file for another, and so on. See `vignette("scripts")` for worked examples of all three and guidance on choosing.

See Also

`tar_target_py_raw()`, `run_py_step()`, `tar_target_jl()`

Examples

```
# Building a target does not run it, so these examples need no Python.
# Inside _targets.R:
# scripts/sum.py:
#   result = sum(x)
# scripts/post.R:
#   py$result
list(
  tarpolyglot::tar_target_py(
    name = py_sum,
    script = "scripts/sum.py",
```

```

    inputs = c(x = "prepared_x"),
    post_script = "scripts/post.R"
  )
)

# The three ways to supply a script (see the "Script options" section):
# 1. Literal path: untracked, editing the file does NOT re-run the step.
tarpolyglot::tar_target_py(
  name = demo_literal, script = "py/step.py", retrieve = "result"
)

# 2. tar_target_path(): tracked, editing the file DOES re-run the step.
list(
  targets::tar_target(step_py, "py/step.py", format = "file"),
  tarpolyglot::tar_target_py(
    name = demo_tracked,
    script = tarpolyglot::tar_target_path("step_py"),
    retrieve = "result"
  )
)

# 3. tar_code(): inline, editing the code DOES re-run the step. A string
#   carries foreign source; an R { } block carries inline R.
tarpolyglot::tar_target_py(
  name = demo_inline,
  script = tarpolyglot::tar_code("result = 1 + 1"),
  post_script = tarpolyglot::tar_code({ py_get("result") })
)

# Tracking a helper module the script imports. Point `inputs` at a
# format = "file" target so editing the helper also re-runs the step;
# `inputs` takes the *target* name, not the path. py/step.py then loads
# the helper from the bound path rather than a hard-coded one:
#   import os, sys
#   sys.path.insert(0, os.path.dirname(helper_path))
#   import helper
#   result = sum(helper.scale(x))
list(
  targets::tar_target(helper_file, "py/helper.py", format = "file"),
  tarpolyglot::tar_target_py(
    name = demo_helper,
    script = "py/step.py",
    inputs = c(x = "prepared_x", helper_path = "helper_file"),
    pre_script = tarpolyglot::tar_code({
      to_py <- list(x = x, helper_path = helper_path)
    }),
    retrieve = "result"
  )
)

```

tar_target_py_raw

Target that runs a Python script (raw / factory form)

Description

Character-based constructor mirroring `targets::tar_target_raw()`: `name` is a string and it is meant for use *inside targets factories*. Returns a single `targets` target whose command runs an optional R pre-script, `script` (via **reticulate**), and an optional R post-script, then returns a converted R object or a character vector of files. See `run_py_step()` for the pre/post-script contract (the `to_py` hand-off and the `py / py_get` helpers). For direct use in `_targets.R`, see `tar_target_py()`. The Julia twin is `tar_target_jl_raw()`.

Usage

```
tar_target_py_raw(
  name,
  script,
  pre_script = NULL,
  post_script = NULL,
  inputs = NULL,
  output = "object",
  retrieve = NULL,
  files = NULL,
  python_version = NULL,
  env = NULL,
  env_manager = "system",
  python = NULL,
  pattern = NULL,
  packages = targets::tar_option_get("packages"),
  library = targets::tar_option_get("library"),
  deps = NULL,
  string = NULL,
  format = NULL,
  repository = targets::tar_option_get("repository"),
  iteration = targets::tar_option_get("iteration"),
  error = targets::tar_option_get("error"),
  memory = targets::tar_option_get("memory"),
  garbage_collection = isTRUE(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"),
  description = targets::tar_option_get("description")
)
```

Arguments

<code>name</code>	Character string, the target name.
<code>script</code>	Path to the Python script to run (required). Accepts a literal path, a <code>tar_target_path()</code>

	reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
<code>pre_script</code>	Optional path to an R script run before the Python script. See <code>run_py_step()</code> ; assign a named list to <code>_py</code> to hand objects to Python. Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
<code>post_script</code>	Optional path to an R script run after the Python script. See <code>run_py_step()</code> ; helpers <code>py</code> and <code>py_get()</code> are available, and its last expression (object mode) or returned paths (file mode) become the value. Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
<code>inputs</code>	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.
<code>output</code>	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").
<code>retrieve</code>	Optional character vector of foreign-session variable names to return when no <code>post_script</code> is supplied in object mode. One name returns that object; several return a named list.
<code>files</code>	Optional character vector of file paths to return when no <code>post_script</code> is supplied in file mode.
<code>python_version</code>	Optional Python version to select (e.g. "3.12" or ">=3.11"), used only when no environment and no explicit python path are given. reticulate fetches/selects it (via its uv-backed ephemeral environment) with <code>reticulate::py_require()</code> . Default NULL uses the computer/global default Python. Use this when you only care about the interpreter <i>version</i> and are happy for reticulate to build a <i>throwaway</i> environment; for a <i>pinned</i> , <i>reproducible</i> environment use <code>env / env_manager</code> (or <code>python</code>) instead.
<code>env, env_manager, python</code>	Python environment selection: the reproducible alternatives to <code>python_version</code> . Use <code>env + env_manager</code> to point at an existing environment built by a known tool, or <code>python</code> for one explicit interpreter path. <code>env_manager</code> is one of "system", "virtualenv", "venv", "conda", "uv", "poetry"; <code>env</code> is the corresponding virtualenv/conda name or path (or poetry project directory); <code>python</code> is an explicit interpreter path. Precedence: <code>python > environment (env/env_manager) > python_version > default</code> . ("virtualenv", "venv" and "uv" all point at a standard virtualenv, including one created by <code>renv::use_python()</code> , and behave identically.) For "virtualenv"/"venv"/"uv"/"poetry", an <code>env</code> that is an already-existing directory (relative to the working directory, or absolute) is resolved to an absolute path before use, so a relative venv path (e.g. ".venv", created with <code>uv venv .venv</code>) works as expected; otherwise <code>reticulate::use_virtualenv()</code> would misread a separator-less relative path as the <i>name</i> of an environment under its own virtualenv root instead of a path on disk. A bare name that does not correspond to an existing directory is passed through unchanged, so a genuine

	named environment (one already registered under that root) still resolves. When python or an environment is given, the selection also takes priority over an ambient RETICULATE_PYTHON environment variable (e.g. one set by the RStudio project Python config and inherited by crew workers), which reticulate would otherwise let silently override the request.
pattern	Optional targets dynamic-branching pattern as a language object (e.g. <code>quote(map(x))</code>), forwarded to <code>targets::tar_target_raw()</code> . The <code>tarpolyglot_map()</code> family is also accepted and behaves identically to the plain targets patterns here (they only differ on the Rust constructor).
packages, library	Character vectors of R packages (and library paths) to load for the target, forwarded to <code>targets::tar_target_raw()</code> .
deps, string	Advanced <code>targets::tar_target_raw()</code> arguments: extra dependency names and a string used for change detection.
format, repository, iteration	Storage/iteration settings forwarded to <code>targets::tar_target_raw()</code> . format defaults to "file" when output = "file", otherwise to the targets option default.
error, memory, garbage_collection, deployment, priority, resources, storage, retrieval, cue, description	Standard <code>targets::tar_target_raw()</code> execution/behaviour arguments, forwarded unchanged.

Details

All `targets::tar_target_raw()` arguments are forwarded unchanged, so dynamic branching (pattern), storage format, deployment, resources, cue, etc. all work as usual. Upstream targets are wired in through inputs, which become dependencies (and are sliced under dynamic branching).

The interpreter/environment selection arguments (python, env, env_manager, python_version) are forwarded to `run_py_step()`, which documents them (they are *alternatives*: normally set only one; precedence python > env/env_manager > python_version > none). See also `vignette("python")` for a decision guide with examples.

Value

A targets target object.

Script options

Every script argument (script, and pre_script / post_script where the constructor has them) accepts the same three forms. The choice is not cosmetic: it decides whether editing the code re-runs the target.

A literal path string e.g. `script = "py/step.py"`. The file is read when the step runs, but it is **not** tracked, so editing it does **not** invalidate the target: targets will happily reuse a stale result until some *other* dependency changes. Simplest form, and a reasonable default once a script has settled.

A `tar_target_path()` reference e.g. `script = tar_target_path("step_py")`, naming an upstream `tar_target(step_py, "py/step.py", format = "file")`. The file becomes a real targets dependency, so editing it **does** invalidate this target and the step re-runs. This is what you want while a script is still changing, and the recommended form for reproducible pipelines.

Inline code via `tar_code()` e.g. `script = tar_code("result = sum(x)")`. The code lives in `_targets.R` rather than in a file, and is embedded in the target's command, so targets hashes it and editing it **does** invalidate the target. A character string carries foreign source (Python, Julia, Rust, C++) or R source; an R `{ ... }` block carries inline R and is accepted only by `pre_script` / `post_script`, never by the foreign script.

Mixing forms in one call is fine: a tracked script with a literal `post_script`, inline code for one and a file for another, and so on. See `vignette("scripts")` for worked examples of all three and guidance on choosing.

See Also

`tar_target_py()`, `run_py_step()`, `tar_target_jl_raw()`

Examples

```
# Building a target does not run it, so these examples need no Python.
# Inside a targets factory:
# scripts/pre.R:
#   to_py <- list(x = x)
# scripts/sum.py:
#   result = sum(x)
# scripts/post.R:
#   py$result
tarpolyglot::tar_target_py_raw(
  name = "py_sum",
  script = "scripts/sum.py",
  inputs = c(x = "prepared_x"),
  pre_script = "scripts/pre.R",
  post_script = "scripts/post.R"
)

# The three ways to supply a script (see the "Script options" section):
# 1. Literal path: untracked, editing the file does NOT re-run the step.
tarpolyglot::tar_target_py_raw(
  name = "demo_literal", script = "py/step.py", retrieve = "result"
)

# 2. tar_target_path(): tracked, editing the file DOES re-run the step.
list(
  targets::tar_target(step_py, "py/step.py", format = "file"),
  tarpolyglot::tar_target_py_raw(
    name = "demo_tracked",
    script = tarpolyglot::tar_target_path("step_py"),
    retrieve = "result"
  )
)
```

```

)

# 3. tar_code(): inline, editing the code DOES re-run the step. A string
#   carries foreign source; an R { } block carries inline R.
tarpolyglot::tar_target_py_raw(
  name = "demo_inline",
  script = tarpolyglot::tar_code("result = 1 + 1"),
  post_script = tarpolyglot::tar_code({ py_get("result") })
)

# Tracking a helper module the script imports. Point `inputs` at a
# format = "file" target so editing the helper also re-runs the step;
# `inputs` takes the *target* name, not the path. py/step.py then loads
# the helper from the bound path rather than a hard-coded one:
#   import os, sys
#   sys.path.insert(0, os.path.dirname(helper_path))
#   import helper
#   result = sum(helper.scale(x))
list(
  targets::tar_target(helper_file, "py/helper.py", format = "file"),
  tarpolyglot::tar_target_py_raw(
    name = "demo_helper",
    script = "py/step.py",
    inputs = c(x = "prepared_x", helper_path = "helper_file"),
    pre_script = tarpolyglot::tar_code({
      to_py <- list(x = x, helper_path = helper_path)
    }),
    retrieve = "result"
  )
)

```

tar_target_rs

Target that runs a Rust script

Description

Non-standard-evaluation constructor mirroring `targets::tar_target()` for Rust: pass a bare name and unquoted pattern, for direct use in `_targets.R`. Compiles the `#[extendr]` functions in script with `rextendr::rust_source()` and calls them from the R `post_script`. Delegates to `tar_target_rs_raw()`; see it and `run_rs_step()` for the full reference. The Python/Julia twins are `tar_target_py()` / `tar_target_jl()`.

Usage

```

tar_target_rs(
  name,
  script,
  post_script = NULL,
  inputs = NULL,

```

```

output = "object",
files = NULL,
dependencies = NULL,
features = NULL,
profile = NULL,
toolchain = NULL,
pattern = NULL,
packages = targets::tar_option_get("packages"),
library = targets::tar_option_get("library"),
deps = NULL,
string = NULL,
format = NULL,
repository = targets::tar_option_get("repository"),
iteration = targets::tar_option_get("iteration"),
error = targets::tar_option_get("error"),
memory = targets::tar_option_get("memory"),
garbage_collection = isTRUE(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"),
description = targets::tar_option_get("description")
)

```

Arguments

name	Symbol, the target name (unquoted).
script	Path to the Rust script with <code>#[extendr]</code> functions (required). Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
post_script	Path to an R script run after compilation, where the compiled functions and inputs are in scope. Its last expression is the value (object mode); it returns file paths (file mode). Required for object mode. Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
inputs	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").
files	Optional character vector of file paths to return when no <code>post_script</code> is supplied in file mode.

dependencies, features, profile	Passed to <code>rextendr::rust_source()</code> : crate dependencies (named list), Cargo features, and build profile.
toolchain	Optional rustup toolchain (e.g. "stable-x86_64-pc-windows-gnu"); sets RUSTUP_TOOLCHAIN for the build.
pattern	Optional branching pattern as described in the targets package documentation , unquoted (e.g. <code>map(x)</code>). The patterns included in the targets package (<code>map()</code> , <code>head()</code> , ...) are accepted, but it is recommended to use the tarpolyglot pattern functions instead, as they compile the library once and reuse it across the branches (see <code>tarpolyglot_map()</code> , <code>tarpolyglot_head()</code> , <code>tarpolyglot_tail()</code> , <code>tarpolyglot_cross()</code> , <code>tarpolyglot_slice()</code> , <code>tarpolyglot_sample()</code>).
packages, library	Character vectors of R packages (and library paths) to load for the target, forwarded to <code>targets::tar_target_raw()</code> .
deps, string	Advanced <code>targets::tar_target_raw()</code> arguments: extra dependency names and a string used for change detection.
format, repository, iteration	Storage/iteration settings forwarded to <code>targets::tar_target_raw()</code> . format defaults to "file" when output = "file", otherwise to the targets option default.
error, memory, garbage_collection, deployment, priority, resources, storage, retrieval, cue, description	Standard <code>targets::tar_target_raw()</code> execution/behaviour arguments, forwarded unchanged.

Value

A targets target object. When pattern uses `tarpolyglot_map()` it is instead a list of two targets: the `<name>_rust_lib` compile target and the branched `<name>` target.

Script options

Every script argument (script, and pre_script / post_script where the constructor has them) accepts the same three forms. The choice is not cosmetic: it decides whether editing the code re-runs the target.

A literal path string e.g. `script = "py/step.py"`. The file is read when the step runs, but it is **not** tracked, so editing it does **not** invalidate the target: targets will happily reuse a stale result until some *other* dependency changes. Simplest form, and a reasonable default once a script has settled.

A `tar_target_path()` reference e.g. `script = tar_target_path("step_py")`, naming an upstream `tar_target(step_py, "py/step.py", format = "file")`. The file becomes a real targets dependency, so editing it **does** invalidate this target and the step re-runs. This is what you want while a script is still changing, and the recommended form for reproducible pipelines.

Inline code via `tar_code()` e.g. `script = tar_code("result = sum(x)")`. The code lives in `_targets.R` rather than in a file, and is embedded in the target's command, so targets hashes

it and editing it **does** invalidate the target. A character string carries foreign source (Python, Julia, Rust, C++) or R source; an R `{ ... }` block carries inline R and is accepted only by `pre_script` / `post_script`, never by the foreign script.

Mixing forms in one call is fine: a tracked script with a literal `post_script`, inline code for one and a file for another, and so on. See `vignette("scripts")` for worked examples of all three and guidance on choosing.

See Also

`tar_target_rs_raw()`, `tarpolyglot_map()`, `run_rs_step()`, `tar_target_py()`, `tar_target_jl()`

Examples

```
# Building a target does not run it, so these examples need no Rust toolchain.
# scripts/square.rs:
#   #[extendr]
#   fn square(x: f64) -> f64 { x * x }
# scripts/post.R:
#   square(x)
list(
  tarpolyglot::tar_target_rs(
    name = rs_square,
    script = "scripts/square.rs",
    inputs = c(x = "value"),
    post_script = "scripts/post.R"
  )
)

# The three ways to supply a script (see the "Script options" section):
# 1. Literal path: untracked, editing the file does NOT re-run the step.
tarpolyglot::tar_target_rs(
  name = demo_literal, script = "rs/step.rs", post_script = "R/post.R"
)

# 2. tar_target_path(): tracked, editing the file DOES re-run the step.
list(
  targets::tar_target(step_rs, "rs/step.rs", format = "file"),
  tarpolyglot::tar_target_rs(
    name = demo_tracked,
    script = tarpolyglot::tar_target_path("step_rs"),
    post_script = "R/post.R"
  )
)

# 3. tar_code(): inline, editing the code DOES re-run the step. A string
#   carries foreign source; an R { } block carries inline R.
tarpolyglot::tar_target_rs(
  name = demo_inline,
  script = tarpolyglot::tar_code("#[extendr] fn one() -> f64 { 1.0 }"),
  post_script = tarpolyglot::tar_code({ one() })
)
)
```

tar_target_rs_raw	<i>Target that runs a Rust script (raw / factory form)</i>
-------------------	--

Description

Character-based constructor mirroring `targets::tar_target_raw()` for Rust, for use inside targets factories. Returns a single targets target whose command compiles the `#[extendr]` functions in script with `rextendr::rust_source()` and then evaluates an R **post-script** that calls those functions (with the upstream inputs in scope) and returns the value. See `run_rs_step()` for the contract. For direct use in `_targets.R`, see `tar_target_rs()`.

Usage

```
tar_target_rs_raw(
  name,
  script,
  post_script = NULL,
  inputs = NULL,
  output = "object",
  files = NULL,
  dependencies = NULL,
  features = NULL,
  profile = NULL,
  toolchain = NULL,
  pattern = NULL,
  packages = targets::tar_option_get("packages"),
  library = targets::tar_option_get("library"),
  deps = NULL,
  string = NULL,
  format = NULL,
  repository = targets::tar_option_get("repository"),
  iteration = targets::tar_option_get("iteration"),
  error = targets::tar_option_get("error"),
  memory = targets::tar_option_get("memory"),
  garbage_collection = isTRUE(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"),
  description = targets::tar_option_get("description")
)
```

Arguments

name	Character string, the target name.
------	------------------------------------

script	Path to the Rust script with <code>#[extendr]</code> functions (required). Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
post_script	Path to an R script run after compilation, where the compiled functions and inputs are in scope. Its last expression is the value (object mode); it returns file paths (file mode). Required for object mode. Accepts a literal path, a <code>tar_target_path()</code> reference, or inline code from <code>tar_code()</code> ; see the "Script options" section below.
inputs	Named character vector (or list) mapping the name seen inside the step (in the R environment and, after the hand-off, in the foreign session) to the name of an upstream target, e.g. <code>c(x = "prepared_x")</code> . Each upstream target becomes a dependency of this target and is bound by that name in the step environment; under dynamic branching the per-branch slice is bound instead.
output	Output mode: "object" (default) returns a converted R object, "file" returns a character vector of file paths (and defaults format to "file").
files	Optional character vector of file paths to return when no <code>post_script</code> is supplied in file mode.
dependencies, features, profile	Passed to <code>rextendr::rust_source()</code> : crate dependencies (named list), Cargo features, and build profile.
toolchain	Optional rustup toolchain (e.g. "stable-x86_64-pc-windows-gnu"); sets <code>RUSTUP_TOOLCHAIN</code> for the build.
pattern	Optional branching pattern as described in the targets package documentation , as a language object (e.g. <code>quote(map(x))</code>), forwarded to <code>targets::tar_target_raw()</code> . The patterns included in the targets package (<code>map()</code> , <code>head()</code> , ...) are accepted, but it is recommended to use the tarpolyglot pattern functions instead, as they compile the library once and reuse it across the branches (see <code>tarpolyglot_map()</code> , <code>tarpolyglot_head()</code> , <code>tarpolyglot_tail()</code> , <code>tarpolyglot_cross()</code> , <code>tarpolyglot_slice()</code> , <code>tarpolyglot_sample()</code>).
packages, library	Character vectors of R packages (and library paths) to load for the target, forwarded to <code>targets::tar_target_raw()</code> .
deps, string	Advanced <code>targets::tar_target_raw()</code> arguments: extra dependency names and a string used for change detection.
format, repository, iteration	Storage/iteration settings forwarded to <code>targets::tar_target_raw()</code> . <code>format</code> defaults to "file" when <code>output = "file"</code> , otherwise to the <code>targets</code> option default.
error, memory, garbage_collection, deployment, priority, resources, storage, retrieval, cue, description	Standard <code>targets::tar_target_raw()</code> execution/behaviour arguments, forwarded unchanged.

Details

Unlike Python/Julia there is **no pre-script** for Rust: inputs are used directly in the post-script alongside the compiled functions. A Rust toolchain and cargo are required; on Windows use the GNU

toolchain (see vignette("rust")).

Under dynamic branching, `pattern = map(...)` recompiles the crate in every branch (Rust has no live interpreter to reuse). Passing a tarpolyglot pattern helper instead (`tarpolyglot_map()`, `tarpolyglot_cross()`, `tarpolyglot_slice()`, `tarpolyglot_head()`, `tarpolyglot_tail()`, `tarpolyglot_sample()`) compiles the crate **once** in a companion target named `<name>_rust_lib` and reuses it across all branches; the constructor then returns *both* targets as a list. See `tarpolyglot_map()`.

Value

A targets target object. When pattern uses `tarpolyglot_map()` it is instead a list of two targets: the `<name>_rust_lib` compile target and the branched `<name>` target.

Script options

Every script argument (`script`, and `pre_script` / `post_script` where the constructor has them) accepts the same three forms. The choice is not cosmetic: it decides whether editing the code re-runs the target.

A literal path string e.g. `script = "py/step.py"`. The file is read when the step runs, but it is **not** tracked, so editing it does **not** invalidate the target: targets will happily reuse a stale result until some *other* dependency changes. Simplest form, and a reasonable default once a script has settled.

A `tar_target_path()` reference e.g. `script = tar_target_path("step_py")`, naming an upstream `tar_target(step_py, "py/step.py", format = "file")`. The file becomes a real targets dependency, so editing it **does** invalidate this target and the step re-runs. This is what you want while a script is still changing, and the recommended form for reproducible pipelines.

Inline code via `tar_code()` e.g. `script = tar_code("result = sum(x)")`. The code lives in `_targets.R` rather than in a file, and is embedded in the target's command, so targets hashes it and editing it **does** invalidate the target. A character string carries foreign source (Python, Julia, Rust, C++) or R source; an R `{ ... }` block carries inline R and is accepted only by `pre_script` / `post_script`, never by the foreign script.

Mixing forms in one call is fine: a tracked script with a literal `post_script`, inline code for one and a file for another, and so on. See vignette("scripts") for worked examples of all three and guidance on choosing.

See Also

`tar_target_rs()`, `tarpolyglot_map()`, `run_rs_step()`, `tar_target_py_raw()`, `tar_target_jl_raw()`

Examples

```
# Building a target does not run it, so these examples need no Rust toolchain.
# scripts/square.rs:
#   #[extendr]
#   fn square(x: f64) -> f64 { x * x }
# scripts/post.R:
#   square(x)
```

```

tarpolyglot::tar_target_rs_raw(
  name = "rs_square",
  script = "scripts/square.rs",
  inputs = c(x = "value"),
  post_script = "scripts/post.R"
)

# The three ways to supply a script (see the "Script options" section):
# 1. Literal path: untracked, editing the file does NOT re-run the step.
tarpolyglot::tar_target_rs_raw(
  name = "demo_literal", script = "rs/step.rs", post_script = "R/post.R"
)

# 2. tar_target_path(): tracked, editing the file DOES re-run the step.
list(
  targets::tar_target(step_rs, "rs/step.rs", format = "file"),
  tarpolyglot::tar_target_rs_raw(
    name = "demo_tracked",
    script = tarpolyglot::tar_target_path("step_rs"),
    post_script = "R/post.R"
  )
)

# 3. tar_code(): inline, editing the code DOES re-run the step. A string
#    carries foreign source; an R { } block carries inline R.
tarpolyglot::tar_target_rs_raw(
  name = "demo_inline",
  script = tarpolyglot::tar_code("#[extendr] fn one() -> f64 { 1.0 }"),
  post_script = tarpolyglot::tar_code({ one() })
)

```

tarpolyglot_cross

Dynamic-branching cross that compiles Rust once

Description

A drop-in replacement for the **targets** `targets::tar_target()` pattern helper `cross()`, for use unquoted in the pattern argument of a tarpolyglot constructor. Like `tarpolyglot_map()`, on `tar_target_rs()` it compiles the extendr crate **once** in a companion `<name>_rust_lib` target and reuses it across every branch (instead of recompiling per branch); on `tar_target_py()` / `tar_target_jl()` it is exactly `cross()`. See `tarpolyglot_map()` for the full explanation and the compile-once mechanics.

Usage

```
tarpolyglot_cross(...)
```

Arguments

... Upstream targets to cross (all combinations), with the same meaning as in the **targets** `cross()` pattern.

Value

This function is a marker consumed by the tarpolyglot constructors and is not meant to be evaluated on its own; calling it directly raises an error.

Only recognised by tarpolyglot constructors

The tarpolyglot_*() pattern helpers work only inside the pattern argument of the tarpolyglot constructors (`tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`, and their `_raw()` forms), which rewrite them to the plain **targets** pattern before building the target. Used directly in a plain `targets::tar_target()` or `targets::tar_target_raw()` they raise an error such as invalid dynamic branching pattern ... Illegal symbols found: tarpolyglot_map, because **targets** validates a pattern against its own fixed set of pattern functions and does not know these helpers. This cannot be fixed from tarpolyglot: **targets** looks its pattern functions up in a locked internal environment, so teaching it a new one would require modifying the **targets** package itself. In a plain **targets** target (which has no foreign code to compile, so nothing to gain) use the native `map()` / `cross()` / `slice()` / `head()` / `tail()` / `sample()` instead.

See Also

The other pattern helpers: `tarpolyglot_map()`, `tarpolyglot_slice()`, `tarpolyglot_head()`, `tarpolyglot_tail()`, `tarpolyglot_sample()`. The constructors that accept them: `tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`

Examples

```
# square.rs:
# #[extendr]
# fn square(x: f64) -> f64 { x * x }
# post.R:
# square(x)
tarpolyglot::tar_target_rs(
  name = rs_grid, script = "square.rs", inputs = c(x = "a", y = "b"),
  post_script = "post.R", pattern = tarpolyglot_cross(a, b)
)
```

tarpolyglot_head

Dynamic-branching head that compiles Rust once

Description

A drop-in replacement for the **targets** `targets::tar_target()` pattern helper `head()`, for use unquoted in the pattern argument of a tarpolyglot constructor. Like `tarpolyglot_map()`, on `tar_target_rs()` it compiles the extendr crate **once** in a companion `<name>_rust_lib` target and reuses it across the kept branches; on `tar_target_py()` / `tar_target_jl()` it is exactly `head()`. See `tarpolyglot_map()` for the full explanation and the compile-once mechanics.

Usage

```
tarpolyglot_head(..., n)
```

Arguments

... Upstream target(s) to branch over, with the same meaning as in the **targets** head() pattern.

n Number of branches to keep from the start, as in the **targets** head() pattern.

Value

This function is a marker consumed by the tarpolyglot constructors and is not meant to be evaluated on its own; calling it directly raises an error.

Only recognised by tarpolyglot constructors

The tarpolyglot_*() pattern helpers work only inside the pattern argument of the tarpolyglot constructors (tar_target_rs(), tar_target_cpp(), tar_target_py(), tar_target_jl(), and their _raw() forms), which rewrite them to the plain **targets** pattern before building the target. Used directly in a plain targets::tar_target() or targets::tar_target_raw() they raise an error such as invalid dynamic branching pattern ... Illegal symbols found: tarpolyglot_map, because **targets** validates a pattern against its own fixed set of pattern functions and does not know these helpers. This cannot be fixed from tarpolyglot: **targets** looks its pattern functions up in a locked internal environment, so teaching it a new one would require modifying the **targets** package itself. In a plain **targets** target (which has no foreign code to compile, so nothing to gain) use the native map() / cross() / slice() / head() / tail() / sample() instead.

See Also

The other pattern helpers: tarpolyglot_map(), tarpolyglot_cross(), tarpolyglot_slice(), tarpolyglot_tail(), tarpolyglot_sample(). The constructors that accept them: tar_target_rs(), tar_target_cpp(), tar_target_py(), tar_target_jl()

Examples

```
# square.rs:
#   #[extendr]
#   fn square(x: f64) -> f64 { x * x }
# post.R:
#   square(x)
tarpolyglot::tar_target_rs(
  name = rs_first, script = "square.rs", inputs = c(x = "vals"),
  post_script = "post.R", pattern = tarpolyglot_head(vals, n = 2)
)
```

tarpolyglot_map

Dynamic-branching map that compiles Rust/C++ once

Description

A drop-in replacement for the **targets** targets::tar_target() pattern helper map(), for use unquoted in the pattern argument of a tarpolyglot constructor (tar_target_rs(), tar_target_cpp(), tar_target_py(), tar_target_jl(), and their _raw forms).

Usage

```
tarpolyglot_map(...)
```

Arguments

... Upstream targets to map over in parallel, exactly as in the **targets** `map()` pattern (e.g. `tarpolyglot_map(x)` or `tarpolyglot_map(x, y)`).

Details

On `tar_target_rs()` and `tar_target_cpp()` it changes how the branches are built: the crate/library is compiled **once** in a companion target named `<name>_rust_lib` / `<name>_cpp_lib`, and every branch reuses that compiled library instead of recompiling. Recompiling per branch is otherwise the default, because Rust and C++ have no live interpreter (contrast Python/Julia, whose interpreter is simply reused). With three branches this turns roughly 3 x compile into 1 x compile + 3 x (near-instant reload).

On `tar_target_py()` and `tar_target_jl()` there is nothing to compile, so `tarpolyglot_map()` is exactly `map()`: it is provided so the same pipeline code can branch every language uniformly.

Use it wherever you would write `map()`: `pattern = tarpolyglot_map(x)`. The arguments are upstream target names to iterate over in parallel, with the same meaning as `targets::tar_target()`'s `map()`.

Value

This function is a marker consumed by the tarpolyglot constructors and is not meant to be evaluated on its own; calling it directly raises an error.

Only recognised by tarpolyglot constructors

The `tarpolyglot_*()` pattern helpers work only inside the pattern argument of the tarpolyglot constructors (`tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`, and their `_raw()` forms), which rewrite them to the plain **targets** pattern before building the target. Used directly in a plain `targets::tar_target()` or `targets::tar_target_raw()` they raise an error such as `invalid dynamic branching pattern ... Illegal symbols found: tarpolyglot_map`, because **targets** validates a pattern against its own fixed set of pattern functions and does not know these helpers. This cannot be fixed from tarpolyglot: **targets** looks its pattern functions up in a locked internal environment, so teaching it a new one would require modifying the **targets** package itself. In a plain **targets** target (which has no foreign code to compile, so nothing to gain) use the native `map()` / `cross()` / `slice()` / `head()` / `tail()` / `sample()` instead.

See Also

The other pattern helpers: `tarpolyglot_cross()`, `tarpolyglot_slice()`, `tarpolyglot_head()`, `tarpolyglot_tail()`, `tarpolyglot_sample()`. The constructors that accept them: `tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`

Examples

```
# rs/square.rs:
#   #[extendr]
#   fn square(x: f64) -> f64 { x * x }
# R/post_square.R:
#   square(x)
list(
  targets::tar_target(vals, c(10, 20, 30)),
  # Rust: compiles rs/square.rs once in `rs_branch_rust_lib`, then squares
  # each branch value reusing that compiled library.
  tarpolyglot::tar_target_rs(
    name = rs_branch,
    script = "rs/square.rs",
    inputs = c(x = "vals"),
    post_script = "R/post_square.R",
    pattern = tarpolyglot_map(vals)
  )
)
```

tarpolyglot_sample	<i>Dynamic-branching sample that compiles Rust once</i>
--------------------	---

Description

A drop-in replacement for the **targets** `targets::tar_target()` pattern helper `sample()`, for use unquoted in the pattern argument of a tarpolyglot constructor. Like `tarpolyglot_map()`, on `tar_target_rs()` it compiles the extendr crate **once** in a companion `<name>_rust_lib` target and reuses it across the sampled branches; on `tar_target_py()` / `tar_target_jl()` it is exactly `sample()`. See `tarpolyglot_map()` for the full explanation and the compile-once mechanics.

Usage

```
tarpolyglot_sample(..., n)
```

Arguments

...	Upstream target(s) to branch over, with the same meaning as in the targets <code>sample()</code> pattern.
n	Number of branches to sample at random, as in the targets <code>sample()</code> pattern.

Value

This function is a marker consumed by the tarpolyglot constructors and is not meant to be evaluated on its own; calling it directly raises an error.

Only recognised by tarpolyglot constructors

The `tarpolyglot_*()` pattern helpers work only inside the pattern argument of the tarpolyglot constructors (`tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`, and their `_raw()` forms), which rewrite them to the plain **targets** pattern before building the target. Used directly in a plain `targets::tar_target()` or `targets::tar_target_raw()` they raise an error such as `invalid dynamic branching pattern ... Illegal symbols found: tarpolyglot_map`, because **targets** validates a pattern against its own fixed set of pattern functions and does not know these helpers. This cannot be fixed from tarpolyglot: **targets** looks its pattern functions up in a locked internal environment, so teaching it a new one would require modifying the **targets** package itself. In a plain **targets** target (which has no foreign code to compile, so nothing to gain) use the native `map()` / `cross()` / `slice()` / `head()` / `tail()` / `sample()` instead.

See Also

The other pattern helpers: `tarpolyglot_map()`, `tarpolyglot_cross()`, `tarpolyglot_slice()`, `tarpolyglot_head()`, `tarpolyglot_tail()`. The constructors that accept them: `tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`

Examples

```
# square.rs:
#   #[extendr]
#   fn square(x: f64) -> f64 { x * x }
# post.R:
#   square(x)
tarpolyglot::tar_target_rs(
  name = rs_rand, script = "square.rs", inputs = c(x = "vals"),
  post_script = "post.R", pattern = tarpolyglot_sample(vals, n = 2)
)
```

tarpolyglot_slice

Dynamic-branching slice that compiles Rust once

Description

A drop-in replacement for the **targets** `targets::tar_target()` pattern helper `slice()`, for use unquoted in the pattern argument of a tarpolyglot constructor. Like `tarpolyglot_map()`, on `tar_target_rs()` it compiles the extendr crate **once** in a companion `<name>_rust_lib` target and reuses it across the selected branches; on `tar_target_py()` / `tar_target_jl()` it is exactly `slice()`. See `tarpolyglot_map()` for the full explanation and the compile-once mechanics.

Usage

```
tarpolyglot_slice(..., index)
```

Arguments

...	Upstream target(s) to branch over, with the same meaning as in the targets <code>slice()</code> pattern.
index	Integer vector of branch indices to keep, as in the targets <code>slice()</code> pattern.

Value

This function is a marker consumed by the tarpolyglot constructors and is not meant to be evaluated on its own; calling it directly raises an error.

Only recognised by tarpolyglot constructors

The `tarpolyglot_*()` pattern helpers work only inside the pattern argument of the tarpolyglot constructors (`tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`, and their `_raw()` forms), which rewrite them to the plain **targets** pattern before building the target. Used directly in a plain `targets::tar_target()` or `targets::tar_target_raw()` they raise an error such as `invalid dynamic branching pattern ... Illegal symbols found: tarpolyglot_map`, because **targets** validates a pattern against its own fixed set of pattern functions and does not know these helpers. This cannot be fixed from tarpolyglot: **targets** looks its pattern functions up in a locked internal environment, so teaching it a new one would require modifying the **targets** package itself. In a plain **targets** target (which has no foreign code to compile, so nothing to gain) use the native `map()` / `cross()` / `slice()` / `head()` / `tail()` / `sample()` instead.

See Also

The other pattern helpers: `tarpolyglot_map()`, `tarpolyglot_cross()`, `tarpolyglot_head()`, `tarpolyglot_tail()`, `tarpolyglot_sample()`. The constructors that accept them: `tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`

Examples

```
# square.rs:
#   #[extendr]
#   fn square(x: f64) -> f64 { x * x }
# post.R:
#   square(x)
tarpolyglot::tar_target_rs(
  name = rs_some, script = "square.rs", inputs = c(x = "vals"),
  post_script = "post.R", pattern = tarpolyglot_slice(vals, index = c(1, 3))
)
```

Description

A drop-in replacement for the **targets** `targets::tar_target()` pattern helper `tail()`, for use unquoted in the pattern argument of a tarpolyglot constructor. Like `tarpolyglot_map()`, on `tar_target_rs()` it compiles the extendr crate **once** in a companion `<name>_rust_lib` target and reuses it across the kept branches; on `tar_target_py()` / `tar_target_jl()` it is exactly `tail()`. See `tarpolyglot_map()` for the full explanation and the compile-once mechanics.

Usage

```
tarpolyglot_tail(..., n)
```

Arguments

<code>...</code>	Upstream target(s) to branch over, with the same meaning as in the targets <code>tail()</code> pattern.
<code>n</code>	Number of branches to keep from the end, as in the targets <code>tail()</code> pattern.

Value

This function is a marker consumed by the tarpolyglot constructors and is not meant to be evaluated on its own; calling it directly raises an error.

Only recognised by tarpolyglot constructors

The `tarpolyglot_*()` pattern helpers work only inside the pattern argument of the tarpolyglot constructors (`tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`, and their `_raw()` forms), which rewrite them to the plain **targets** pattern before building the target. Used directly in a plain `targets::tar_target()` or `targets::tar_target_raw()` they raise an error such as `invalid dynamic branching pattern ... Illegal symbols found: tarpolyglot_map`, because **targets** validates a pattern against its own fixed set of pattern functions and does not know these helpers. This cannot be fixed from tarpolyglot: **targets** looks its pattern functions up in a locked internal environment, so teaching it a new one would require modifying the **targets** package itself. In a plain **targets** target (which has no foreign code to compile, so nothing to gain) use the native `map()` / `cross()` / `slice()` / `head()` / `tail()` / `sample()` instead.

See Also

The other pattern helpers: `tarpolyglot_map()`, `tarpolyglot_cross()`, `tarpolyglot_slice()`, `tarpolyglot_head()`, `tarpolyglot_sample()`. The constructors that accept them: `tar_target_rs()`, `tar_target_cpp()`, `tar_target_py()`, `tar_target_jl()`

Examples

```
# square.rs:
#   #[extendr]
#   fn square(x: f64) -> f64 { x * x }
# post.R:
#   square(x)
tarpolyglot::tar_target_rs(
```

```

name = rs_last, script = "square.rs", inputs = c(x = "vals"),
post_script = "post.R", pattern = tarpolyglot_tail(vals, n = 2)
)

```

toolchain_check

Diagnose the Python, Julia, Rust, and C++ toolchains

Description

Runs a battery of checks for whichever toolchains you ask for and reports the result with `cli` as it goes: interpreter/compiler discovery, environment-manager availability, and, most importantly, whether each toolchain is actually *reachable from a fresh worker process* – the same kind of process crew spawns to run a `tar_target_py()`/`tar_target_jl()`/`tar_target_rs()` step. This is meant to preempt the common "it doesn't run on my machine" class of issue before you find out the hard way, mid-pipeline.

Usage

```

toolchain_check(
  toolchains = c("py", "jl", "rs", "cpp"),
  deep = TRUE,
  quiet = FALSE
)

```

Arguments

<code>toolchains</code>	Character vector, which toolchains to check: any subset of "py", "jl", "rs", "cpp". Default checks all four.
<code>deep</code>	Logical. When TRUE (the default), the Rust and C++ checks additionally compile a trivial function (<code>#[extendr] / / / [[Rcpp::export]]</code>) in a fresh worker to prove the full toolchain actually links together end to end; this is the most informative check for each but also the slowest (well under a minute, but not instant). Set <code>deep = FALSE</code> to skip it and rely on the faster presence checks alone. Python and Julia's fresh-worker checks are cheap either way and always run.
<code>quiet</code>	Logical, default FALSE. Suppress the live <code>cli</code> progress/result output; the return value is unaffected either way.

Details

Environment managers checked per language: for Python, `uv`, `poetry`, and `conda` (three separate, competing tools, plus the `stdlib venv` module on the resolved interpreter). For Julia, `juliaup` (which manages *versions*, checked via presence + the list of installed versions) and, separately, Julia's own `Pkg` project-environment mechanism (which manages *packages*, activated with `Pkg.activate()` – the actual mechanism behind `julia_project/julia_packages`): a fresh worker actually activates a throwaway project to prove it works, not just that Julia is present. Rust has no separate environment concept (see `vignette("rust")`); `rustup/cargo` (on Windows) the GNU toolchain and `Rtools` are checked instead.

Every check that would bind an interpreter (Python, Julia) or compile code (Rust, C++) runs inside a disposable `callr::r()` subprocess, never in your current R session: this means `toolchain_check()` cannot leave `reticulate` or `JuliaCall` bound afterwards, and it is answering the *real* question ("would a crew worker be able to do this right now"), not just "is something already loaded in this session". Plain presence checks (`Sys.which()` for `uv`, `poetry`, `rustup`, and so on, or `requireNamespace()` for `Rcpp`) run directly, since they have no side effects to isolate.

When more than one version is found, each language also reports every installed version it can enumerate, with its path and which one is the default (the one a step would actually use if you set nothing): Python via `uv python list --only-installed` (matched against the resolved interpreter primarily by exact version, since `reticulate`'s default is often an *ephemeral* `uv`-provisioned interpreter that lives under `uv`'s cache rather than its "installed" registry, so the two rarely share a literal path even when they are, in fact, the same build); Julia via the `juliaup` depot; Rust via `rustup toolchain list -v` (which marks its own default inline, no cross-referencing needed). With only one version found, this is a single summary line instead. C++ has no separate version-manager concept to enumerate this way (see `tar_target_cpp()`): it compiles via R's own configured toolchain, so its checks are presence (`Rcpp`; the OS-appropriate compiler, `Rtools` on Windows or the actual R CMD `config CXX` compiler command on macOS/Linux, queried rather than guessed so it is correct under `gcc` or `clang` alike) plus, with `deep = TRUE`, a compile-reachability check. Extension packages such as `RcppArmadillo`/`RcppEigen`/`RcppParallel` are not checked here since `tarpolyglot` itself does not depend on any of them; they are entirely the caller's own choice via `depends` (see `tar_target_cpp()`).

Value

Invisibly, a `data.frame` with one row per check and columns `toolchain` ("`py`"/"`j1`"/"`rs`"/"`cpp`"), `check` (a short label), `status` ("`ok`", "`warn`", or "`fail`"), and `detail` (a human-readable description, e.g. the resolved path and version, or an error message).

See Also

`tar_target_py()`, `tar_target_j1()`, `tar_target_rs()`, `tar_target_cpp()`

Examples

```
# Every check runs in a fresh callr subprocess, and with deep = TRUE it also
# compiles a trivial function, so this is gated on TARPOLYGLOT_EXAMPLES=true
# rather than run on machines with no toolchain installed.
if (identical(Sys.getenv("TARPOLYGLOT_EXAMPLES"), "true")) {
  toolchain_check("py")           # Python only
  toolchain_check(c("j1", "rs")) # Julia and Rust
  toolchain_check("rs", deep = FALSE) # skip the Rust compile test

  res <- toolchain_check(quiet = TRUE) # no console output, just the data.frame
  res[res$status != "ok", ]           # anything that needs attention
}
```

Index

`callr::r()`, 57
`crew::crew_controller_local()`, 3

`polyglot_controller`, 3
`polyglot_controller()`, 5, 8, 10, 15, 16

`Rcpp::sourceCpp()`, 4, 16, 20
`reticulate::py_require()`, 10, 34, 38
`rextendr::rust_source()`, 11, 12, 41, 43, 45, 46
`run_cpp_step`, 4
`run_cpp_step()`, 16, 18–22
`run_jl_step`, 6
`run_jl_step()`, 6, 11, 13, 15, 23–30
`run_py_step`, 9
`run_py_step()`, 6, 8, 13, 15, 32, 33, 35, 37–40
`run_rs_step`, 11
`run_rs_step()`, 6, 41, 44, 45, 47

`tar_code`, 13
`tar_code()`, 4, 5, 7–10, 12, 17, 18, 21, 22, 24, 26, 28, 30, 33, 35, 38, 40, 42, 43, 46, 47
`tar_polyglot_log`, 15
`tar_polyglot_log()`, 3–5, 8, 10
`tar_target_cpp`, 16
`tar_target_cpp()`, 4, 6, 15, 20, 22, 49–51, 53–55, 57
`tar_target_cpp_raw`, 20
`tar_target_cpp_raw()`, 16, 19
`tar_target_jl`, 23
`tar_target_jl()`, 4, 6, 8, 14, 16, 19, 27, 30–32, 35, 41, 44, 48–55, 57
`tar_target_jl_raw`, 27
`tar_target_jl_raw()`, 22, 23, 26, 37, 40, 47
`tar_target_path`, 31
`tar_target_path()`, 5, 6, 8, 11, 13, 14, 17, 18, 21, 22, 24, 26, 28, 30, 33, 35, 37, 38, 40, 42, 43, 46, 47
`tar_target_py`, 32
`tar_target_py()`, 4, 5, 8, 9, 11, 13, 14, 16, 19, 23, 26, 31, 32, 37, 40, 41, 44, 48–55, 57
`tar_target_py_raw`, 36
`tar_target_py_raw()`, 22, 27, 30, 32, 35, 47
`tar_target_rs`, 41
`tar_target_rs()`, 11, 13–16, 19, 21, 31, 32, 45, 47–55, 57
`tar_target_rs_raw`, 45
`tar_target_rs_raw()`, 22, 41, 44
`targets::tar_option_set()`, 3
`targets::tar_target()`, 16, 23, 32, 41, 48–55
`targets::tar_target_raw()`, 18, 20, 21, 25, 27, 29, 34, 35, 37, 39, 43, 45, 46, 49–51, 53–55
`tarpolyglot_cross`, 48
`tarpolyglot_cross()`, 18, 21, 22, 43, 46, 47, 50, 51, 53–55
`tarpolyglot_head`, 49
`tarpolyglot_head()`, 18, 21, 22, 43, 46, 47, 49, 51, 53–55
`tarpolyglot_map`, 50
`tarpolyglot_map()`, 18, 19, 21, 22, 29, 39, 43, 44, 46–50, 52–55
`tarpolyglot_sample`, 52
`tarpolyglot_sample()`, 18, 21, 22, 43, 46, 47, 49–51, 54, 55
`tarpolyglot_slice`, 53
`tarpolyglot_slice()`, 18, 21, 22, 43, 46, 47, 49–51, 53, 55
`tarpolyglot_tail`, 54
`tarpolyglot_tail()`, 18, 21, 22, 43, 46, 47, 49–51, 53, 54
`toolchain_check`, 56