Package: progressr (via r-universe)

November 16, 2026

Version 1.1.0-9004

Title An Inclusive, Unifying API for Progress Updates

Description A minimal, unifying API for scripts and packages to report progress updates from anywhere including when using parallel processing. The package is designed such that the developer can to focus on what progress should be reported on without having to worry about how to present it. The end user has full control of how, where, and when to render these progress updates, e.g. in the terminal using utils::txtProgressBar(), cli::cli_progress_bar(), in a graphical user interface using utils::winProgressBar(), tcltk::tkProgressBar() or shiny::withProgress(), via the speakers using beepr::beep(), or on a file system via the size of a file. Anyone can add additional, customized, progression handlers. The 'progressr' package uses R's condition framework for signaling progress updated. Because of this, progress can be reported from almost anywhere in R, e.g. from classical for and while loops, from map-reduce API:s like the lapply() family of functions, 'purrr', 'plyr', and 'foreach'. It will also work with parallel processing via the 'future' framework, e.g. 'lapply(...) |> futurize()' and 'purrr::map(...) |> futurize()', which uses future.apply::future_lapply() and furrr::future_map() internally. The package is compatible with Shiny applications.

License Apache License (>= 2)

Depends R (>= 3.5.0)

Imports tools, utils

Suggests graphics, tcltk, beepr, cli, crayon, digest, pbmcapply, progress, purrr, foreach, plyr, doFuture, future, future.apply, furrr, ntfy, RPushbullet, rstudioapi, shiny, commonmark, base64enc

VignetteBuilder progressr

Language en-US

Encoding UTF-8

URL https://progressr.futureverse.org,
https://github.com/futureverse/progressr

BugReports https://github.com/futureverse/progressr/issues

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.1.1

Repository https://futureverse-devel.r-universe.dev

Date/Publication 2026-11-16 05:18:11 UTC

RemoteUrl https://github.com/futureverse/progressr

RemoteRef develop

RemoteSha 142409f20f02398e66eb17004d7b871e9f40542a

Contents

handler_ascii_alert . . . 3
handler_beepr . . . 4
handler_brrr . . . 5
handler_callback . . . 6
handler_cli . . . 7
handler_debug . . . 9
handler_filesize . . . 10
handler_notifier . . . 11
handler_ntfy . . . 12
handler_pbcol . . . 13
handler_pbmcapply . . . 14
handler_progress . . . 16
handler_rpushbullet . . . 18
handler_rstudio . . . 19
handler_slowdown . . . 20
handler_terminal_title . . . 21
handler_tkprogressbar . . . 23
handler_tutorial . . . 24
handler_txtprogressbar . . . 25
handler_void . . . 26
handler_winprogressbar . . . 27
handlers . . . 28
progress_progressr . . . 30
progressor . . . 31
progressr . . . 33
progressr.options . . . 35
with_progress . . . 37
withProgressShiny . . . 40

Index **43**

handler_ascii_alert *Progression Handler: Progress Reported as ASCII BEL Symbols (Audio or Blink) in the Terminal*

Description

A progression handler based on `cat("\a", file=stderr())`.

Usage

```
handler_ascii_alert(
  symbol = "\a",
  file = stderr(),
  intrusiveness = getOption("progressr.intrusiveness.audio", 5),
  target = c("terminal", "audio"),
  ...
)
```

Arguments

`symbol`	(character string) The character symbol to be outputted, which by default is the ASCII BEL character (`'\a' = '\007'`) character.
`file`	(connection) A base::connection to where output should be sent.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`...`	Additional arguments passed to `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Examples

```
handlers("ascii_alert")
with_progress({ y <- slow_sum_p(1:10) })
print(y)
```

handler_beepr *Progression Handler: Progress Reported as 'beepr' Sounds (Audio)*

Description

A progression handler for `beepr::beep()`.

Usage

```
handler_beepr(
  initiate = 2L,
  update = 10L,
  finish = 11L,
  interrupt = 9L,
  intrusiveness = getOption("progressr.intrusiveness.audio", 5),
  target = "audio",
  ...
)
```

Arguments

`initiate`, `update`, `finish`, `interrupt`	(integer) Indices of `beepr::beep()` sounds to play when progress starts, is updated, completes, or is interrupted. For silence, use `NA_integer_`.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`...`	Additional arguments passed to `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **beepr** package.

Examples

```
if (requireNamespace("beepr", quietly = TRUE)) {

  handlers("beepr")
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)

}
```

handler_brrr *Progression Handler: Progress Reported as 'BRRR' Sounds (Audio)*

Description

A progression handler for `skrrrahh()` of the **BRRR** package.

Usage

```
handler_brrr(
  initiate = "takeoff",
  update = "bigsean4",
  finish = "khaled3",
  interrupt = "busta",
  intrusiveness = getOption("progressr.intrusiveness.audio", 5),
  target = "audio",
  ...
)
```

Arguments

`initiate`, `update`, `finish`, `interrupt`	(integer or character) Indices or names of `skrrrahh()` sounds to play when progress starts, is updated, completes, or is interrupted. For silence, use `NA_integer_`.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`...`	Additional arguments passed to `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **BRRR** package, which is only available from https://github.com/brooke-watson/BRRR. This can be installed as `remotes::install_github("brooke-watson/BRRR")`.

Examples

```
pkg <- "BRRR"
if (requireNamespace(pkg, quietly = TRUE)) {

  handlers("brrr")
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)
```

```
}
```

`handler_callback` *Progression Handler: Progress Reported via a User-Defined Hook Function*

Description

A progression handler that calls a custom hook function every time this handler receives a progression update, e.g. to forward progress information to logging, GUIs, or other external systems not supported by **progressr**.

Usage

```
handler_callback(
  fcn = NULL,
  intrusiveness = getOption("progressr.intrusiveness.callback", 1),
  target = "callback",
  ...
)
```

Arguments

`fcn`	(function) A hook function that is called whenever this handler reports on an `"initiate"`, `"update"`, or `"finish"` event. It is called as `fcn(type, step, max_steps, message, amount)`, where `type` (character string) is one of `"initiate"`, `"update"`, or `"finish"`, `step` (integer) is the number of steps completed so far, `max_steps` (integer) is the total number of steps, `message` (character string) is the latest progress message, and `amount` (numeric) is the increment in `step` since the previous call. The function should accept `...` to remain forward compatible with future arguments. Any value returned by `fcn` is ignored. If `NULL` (default), no hook function is called, i.e. this handler produces no output.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`...`	Additional arguments passed to `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Examples

```
handlers(handler_callback(fcn = function(type, step, max_steps, message, amount, ...) {
  message(sprintf("[%s] step %d/%d (%+g): %s", type, step, max_steps, amount, message))
}))
with_progress({ y <- slow_sum_p(1:10) })
print(y)
```

---

handler_cli *Progression Handler: Progress Reported via 'cli' Progress Bars (Text) in the Terminal*

Description

A progression handler for `cli::cli_progress_bar()`.

Usage

```
handler_cli(
  show_after = 0,
  intrusiveness = getOption("progressr.intrusiveness.terminal", 1),
  target = "terminal",
  type = c("default", "steps", "percent", "time"),
  ...
)
```

Arguments

`show_after`	(numeric) Number of seconds to wait before displaying the progress bar.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`type`	(character) The type of progress bar to display, which controls the default `format` and `format_done` passed to `cli::cli_progress_bar()`. If `"default"`, the **cli** default format is used (`format = NULL`). If `"steps"`, the progress bar shows the current and total number of steps using the format string `"{cli::pb_spin} {cli::pb_bar} {cli::pb_current}/{cli::pb_total} {cli::pb_status}"` (and `format_done = "{cli::pb_bar} {cli::pb_current}/{cli::pb_total} {cli::pb_status}"`). If `"percent"`, the progress bar shows the percentage completed using the format string `"{cli::pb_spin} {cli::pb_bar} {cli::pb_percent} {cli::pb_status}"` (and `format_done = "{cli::pb_bar} {cli::pb_percent} {cli::pb_status}"`). If `"time"`, the progress bar shows the percentage completed, the current and total number of steps, the estimated time remaining (ETA), and the total elapsed time using the format string `"[{cli::pb_elapsed}] {cli::pb_spin} {cli::pb_bar} {cli::pb_percent} [{cli::pb_current}/{cli::pb_total}] (ETA: {cli::pb_eta}) {cli::pb_status}"` (and `format_done = "[{cli::pb_elapsed}] {cli::pb_bar} {cli::pb_percent} [{cli::pb_current}/{cli::pb_total}]`

{cli::pb_status}"). For the meaning of these format variables, see Progress bar variables in the **cli** package. This argument is ignored if `format` is explicitly specified via `...`.

`...` Additional arguments passed to `cli::cli_progress_bar()` and `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **cli** package.

Appearance

Below are a few examples on how to use and customize this progress handler. In all cases, we use `handlers(global = TRUE)`.

```
library(progressr)
handlers("cli")
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_cli(type = "steps"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_cli(type = "percent"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_cli(type = "time"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_cli(format = "{cli::pb_spin} {cli::pb_bar} {cli::pb_current}/{cli::pb_total} {cli::p
y <- slow_sum_p(1:25)
```

Examples

```
if (requireNamespace("cli", quietly = TRUE)) {
 handlers(handler_cli(format = "{cli::pb_spin} {cli::pb_bar} {cli::pb_percent} {cli::pb_status}"))
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)
}
```

handler_debug *Progression Handler: Progress Reported as Debug Information (Text) in the Terminal*

Description

Progression Handler: Progress Reported as Debug Information (Text) in the Terminal

Usage

```
handler_debug(
  interval = getOption("progressr.interval", 0),
  intrusiveness = getOption("progressr.intrusiveness.debug", 0),
  target = "terminal",
  uuid = FALSE,
  ...
)
```

Arguments

`interval`	(numeric) The minimum time (in seconds) between successive progression updates from this handler.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`uuid`	If TRUE, then the progressor UUID and the owner UUID are shown, otherwise not (default).
`...`	Additional arguments passed to `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Appearance

Below is how this progress handler renders by default at 0%, 30% and 99% progress:

With `handlers(handler_debug())`:

```
[21:27:11.236] (0.000s => +0.001s) initiate: 0/100 (+0) '' {clear=TRUE, enabled=TRUE, status=}
[21:27:11.237] (0.001s => +0.000s) update: 0/100 (+0) 'Starting' {clear=TRUE, enabled=TRUE, status=}
[21:27:14.240] (3.004s => +0.002s) update: 30/100 (+30) 'Importing' {clear=TRUE, enabled=TRUE, status=
[21:27:16.245] (5.009s => +0.001s) update: 100/100 (+70) 'Summarizing' {clear=TRUE, enabled=TRUE, statu
[21:27:16.246] (5.010s => +0.003s) update: 100/100 (+0) 'Summarizing' {clear=TRIE, enabled=TRUE, status
```

Examples

```
handlers("debug")
with_progress({ y <- slow_sum_p(1:10) })
print(y)
```

`handler_filesize` *Progression Handler: Progress Reported as the Size of a File on the File System*

Description

Progression Handler: Progress Reported as the Size of a File on the File System

Usage

```
handler_filesize(
  file = "default.progress",
  intrusiveness = getOption("progressr.intrusiveness.file", 5),
  target = "file",
  enable = getOption("progressr.enable", TRUE),
  ...
)
```

Arguments

`file`	(character) A filename.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`enable`	(logical) If FALSE, then progress is not reported.
`...`	Additional arguments passed to `make_progression_handler()`.

Details

This progression handler reports progress by updating the size of a file on the file system. This provides a convenient way for an R script running in batch mode to report on the progress such that the user can peek at the file size (by default in 0-100 bytes) to assess the amount of the progress made, e.g. `ls -l -- *.progress`. If the '`*.progress`' file is accessible via for instance SSH, SFTP, FTPS, HTTPS, etc., then progress can be assessed from a remote location.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Examples

```
## Not run:
handlers(handler_filesize(file = "myscript.progress"))
with_progress(y <- slow_sum_p(1:100))
print(y)

## End(Not run)
```

`handler_notifier` *Progression Handler: Progress Reported via the Operating-System Notification Framework (GUI, Text)*

Description

A progression handler for `notify()` of the **notifier** package.

Usage

```
handler_notifier(
  intrusiveness = getOption("progressr.intrusiveness.notifier", 10),
  target = "gui",
  ...
)
```

Arguments

`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`...`	Additional arguments passed to `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **notifier** package, which is only available from https://github.com/gaborcsardi/notifier. This can be installed as `remotes::install_github("gaborcsardi/notifier@62c`

Examples

```
pkg <- "notifier"
if (requireNamespace(pkg, quietly = TRUE)) {

  handlers("notifier")
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)

}
```

---

handler_ntfy *Progression Handler: Progress Reported via the Ntfy.sh Messaging Service*

Description

A progression handler for `ntfy::ntfy_send()` of the **ntfy** package, which sends notifications via the https://ntfy.sh framework.

Usage

```
handler_ntfy(
  intrusiveness = getOption("progressr.intrusiveness.ntfy", 5),
  target = "gui",
  ...,
  title = "Progress update from R"
)
```

Arguments

`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`title`	title of notification. See https://docs.ntfy.sh/publish/#message-title
`...`	Additional arguments passed to `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **ntfy** package.

Examples

```
pkg <- "ntfy"
if (requireNamespace(pkg, quietly = TRUE)) {
  ## We need to specify a ntfy.sh topic that progress messages
  ## should be sent to. See help("ntfy_topic", package = "ntfy")
  ## for details
  Sys.setenv(NTFY_TOPIC = "R-my-secret-topic")

  handlers("ntfy")
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)
}
```

---

handler_pbcol *Progression Handler: Progress Reported as an ANSI Background Color in the Terminal*

Description

Progression Handler: Progress Reported as an ANSI Background Color in the Terminal

Usage

```
handler_pbcol(
  adjust = 0,
  pad = 1L,
  complete = function(s) cli::bg_blue(cli::col_white(s)),
  incomplete = function(s) cli::bg_cyan(cli::col_white(s)),
  intrusiveness = getOption("progressr.intrusiveness.terminal", 1),
  target = "terminal",
  ...
)
```

Arguments

`adjust`	(numeric) The adjustment of the progress update, where `adjust = 0` positions the message to the very left, and `adjust = 1` positions the message to the very right.
`pad`	(integer) Amount of padding on each side of the message, where padding is done by spaces.
`complete, incomplete`	(function) Functions that take "complete" and "incomplete" strings that comprise the progress bar as input and annotate them to reflect their two different parts. The default is to annotation them with two different background colors and the same foreground color using the **cli** package.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.

target	(character vector) Specifies where progression updates are rendered.
...	Additional arguments passed to make_progression_handler().

Value

A function of class progression_handler that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **cli** package.

Appearance

Below are a few examples on how to use and customize this progress handler. In all cases, we use handlers(global = TRUE).

```
library(progressr)
handlers("pbcol")
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_pbcol(adjust = 0.5))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_pbcol(
  adjust = 1,
  complete = function(s) cli::bg_red(cli::col_black(s)),
  incomplete = function(s) cli::bg_cyan(cli::col_black(s))
))
y <- slow_sum_p(1:25)
```

Examples

```
handlers(handler_pbcol)
with_progress({ y <- slow_sum_p(1:10) })
print(y)
```

---

handler_pbmcapply *Progression Handler: Progress Reported via 'pbmcapply' Progress Bars (Text) in the Terminal*

Description

A progression handler for pbmcapply::progressBar().

Usage

```
handler_pbmcapply(
  char = "=",
  substyle = 3L,
  style = "ETA",
  file = stderr(),
  intrusiveness = getOption("progressr.intrusiveness.terminal", 1),
  target = "terminal",
  ...
)
```

Arguments

`char`	(character) The symbols to form the progress bar for `utils::txtProgressBar()`.
`substyle`	(integer) The progress-bar substyle according to `pbmcapply::progressBar()`.
`style`	(character) The progress-bar style according to
`file`	(connection) A base::connection to where output should be sent.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`...`	Additional arguments passed to `pbmcapply::progressBar()` and `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **pbmcapply** package.

Appearance

Below are a few examples on how to use and customize this progress handler. In all cases, we use `handlers(global = TRUE)`. Since `style = "txt"` corresponds to using `handler_txtprogressbar()` with `style = substyle`, the main usage of this handler is with `style = "ETA"` (default) for which `substyle` is ignored.

```
library(progressr)
handlers("pbmcapply")
y <- slow_sum_p(1:25)
```

Examples

```
if (requireNamespace("pbmcapply", quietly = TRUE)) {

  handlers("pbmcapply")
```

```
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)

}
```

---

handler_progress *Progression Handler: Progress Reported via 'progress' Progress Bars (Text) in the Terminal*

Description

A progression handler for `progress::progress_bar()`.

Usage

```
handler_progress(
  format = NULL,
  show_after = 0,
  intrusiveness = getOption("progressr.intrusiveness.terminal", 1),
  target = "terminal",
  type = c("default", "steps", "percent", "time"),
  ...
)
```

Arguments

`format`	(character string) The format of the progress bar. If `NULL`, the format is determined by the `type` argument.
`show_after`	(numeric) Number of seconds to wait before displaying the progress bar.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`type`	(character) The type of progress bar to display, which controls the default `format` passed to `progress::progress_bar()`. If `"default"`, the format string is `":spin [:bar] :percent :message"`. If `"steps"`, the format string is `":spin [:bar] :current/:total :message"`. If `"percent"`, the format string is `":spin [:bar] :percent :message"`. If `"time"`, the format string is `"[:elapsed] :spin [:bar] :percent [:current/:total] (ETA: :eta) :message"`. For the meaning of these format variables, see progress::progress_bar. This argument is ignored if `format` is explicitly specified.
`...`	Additional arguments passed to `progress::progress_bar()` and `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **progress** package.

Appearance

Below are a few examples on how to use and customize this progress handler. In all cases, we use `handlers(global = TRUE)`.

```
library(progressr)
handlers("progress")
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_progress(complete = "#"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_progress(type = "steps"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_progress(type = "percent"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_progress(type = "time"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_progress(format = ":spin [:bar] :percent :message"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_progress(format = ":percent [:bar] :eta :message"))
y <- slow_sum_p(1:25)
```

Examples

```
if (requireNamespace("progress", quietly = TRUE)) {

  handlers(handler_progress(format = ":spin [:bar] :percent :message"))
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)

}
```

`handler_rpushbullet` *Progression Handler: Progress Reported via the Pushbullet Messaging Service*

Description

A progression handler for `RPushbullet::pbPost()` of the **RPushbullet** package.

Usage

```
handler_rpushbullet(
  intrusiveness = getOption("progressr.intrusiveness.rpushbullet", 5),
  target = "gui",
  ...,
  title = "Progress update from R",
  recipients = NULL,
  email = NULL,
  channel = NULL,
  apikey = NULL,
  devices = NULL
)
```

Arguments

`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`title`	The title of the note being posted.
`recipients`	A character or numeric vector indicating the devices this post should go to. If missing, the default device is looked up from an optional setting, and if none has been set the push is sent to all devices.
`email`	An alternative way to specify a recipient is to specify an email address. If both `recipients` and `email` are present, `recipients` is used.
`channel`	A channel tag used to specify the name of the channel as the recipient. If either `recipients` or `email` are present, they will take precedence over `channel`.
`apikey`	The API key used to access the service. It can be supplied as an argument here, via the global option `rpushbullet.key`, or via the file `~/.rpushbullet.json` which is read at package initialization (and, if found, also sets the global option).
`devices`	The device to which this post is pushed. It can be supplied as an argument here, or via the file `~/.rpushbullet.json` which is read at package initialization.
`...`	Additional arguments passed to `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **RPushbullet** package, a Pushbullet account, and configuration according to the instructions of the **RPushbullet** package. It also requires internet access from the computer where this progress handler is registered.

Examples

```
pkg <- "RPushbullet"
if (requireNamespace(pkg, quietly = TRUE)) {

  handlers("rpushbullet")
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)

}
```

---

handler_rstudio *Progression Handler: Progress Reported in the RStudio Console*

Description

Progression Handler: Progress Reported in the RStudio Console

Usage

```
handler_rstudio(
  intrusiveness = getOption("progressr.intrusiveness.gui", 1),
  target = "gui",
  title = function() format(Sys.time(), "Console %X"),
  ...
)
```

Arguments

`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`title`	(character or a function) The "name" of the progressor, which is displayed in front of the progress bar. If a function, then the name is created dynamically by calling the function when the progressor is created.
`...`	Additional arguments passed to `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler works only in the RStudio Console.

Use this progression handler by default

To use this handler by default whenever using the RStudio Console, add the following to your '`~/.Rprofile`' startup file:

```
if (requireNamespace("progressr", quietly = TRUE)) {
  if (Sys.getenv("RSTUDIO") == "1" && !nzchar(Sys.getenv("RSTUDIO_TERM"))) {
    options(progressr.handlers = progressr::handler_rstudio)
  }
}
```

Examples

```
if (requireNamespace("rstudioapi", quietly = TRUE) && rstudioapi::isAvailable()) {

  handlers("rstudio")
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)

}
```

---

handler_slowdown *Progression Handler: Slow Down Progress Reporting*

Description

Progression Handler: Slow Down Progress Reporting

Usage

```
handler_slowdown(
  duration = 10,
  intrusiveness = 0,
  target = "void",
  enable = getOption("progressr.enable", TRUE),
  ...
)
```

Arguments

`duration`	(numeric) Added slowdown (in seconds) across all progress steps, e.g. if `duration = 10.0` and `steps = 100L`, then each step unit is delayed 0.1 seconds, totaling a ten-second slowdown.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.

target	(character vector) Specifies where progression updates are rendered.
enable	(logical) If FALSE, then progress is not reported.
...	Additional arguments passed to make_progression_handler().

Details

This progression handler delays each progress step a certain number of seconds. It does not render any output.

Value

A function of class progression_handler that takes a progression condition as its first and only argument.

Examples

```
## Not run:
handlers(handler_slowdown(duration = 30.0)) ## 30-second total slowdown
with_progress(y <- slow_sum_p(1:100))
print(y)

## End(Not run)
```

---

handler_terminal_title

Progression Handler: Progress Reported in the Terminal Title

Description

A progression handler that reports on progress in the title of the terminal window or tab, e.g. [42%] 21/50 | ETA 00:12 | message. This is useful when the terminal is in the background, or when the console is busy with other output.

Usage

```
handler_terminal_title(
  spinner = NULL,
  file = stderr(),
  intrusiveness = getOption("progressr.intrusiveness.gui", 1),
  target = "gui",
  ...
)
```

Arguments

spinner	(character vector) Symbols to cycle through on each progress update, including updates with `amount = 0`, which signal that the process is still alive. If `NULL`, no spinner is shown. The default is a set of Unicode braille symbols, if the current locale supports UTF-8, otherwise `-`, `\`, `|`, and `/`.
file	(connection) A base::connection to where the escape sequences that set the terminal title should be sent.
intrusiveness	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
target	(character vector) Specifies where progression updates are rendered.
...	Additional arguments passed to `make_progression_handler()`.

Details

The terminal title is set using the xterm "OSC 0" escape sequence, which is supported by most modern terminals, e.g. xterm, GNOME Terminal, Konsole, macOS Terminal, iTerm2, and Windows Terminal. Inside `tmux` and GNU `screen`, it sets the title of the pane or window, which is displayed by the outer terminal only if the multiplexer is configured to do so, e.g. `set-option -g set-titles on` for `tmux`.

When progress completes, or is interrupted, the terminal title is restored using the xterm title stack. Terminals that do not support the title stack are instead reset to an empty title, which most terminals display as their default title.

This handler outputs nothing, if `file` is not a terminal, if the terminal is a "dumb" terminal, or if R runs in the RStudio Console or in Emacs. On MS Windows, it outputs only when R runs in a terminal known to support escape sequences, e.g. Windows Terminal.

Since the terminal title does not interfere with the output in the console, this handler can be combined with other progression handlers, e.g. `handlers(c("txtprogressbar", "terminal_title"))`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Examples

```
handlers("terminal_title")
with_progress({ y <- slow_sum_p(1:10) })
print(y)

## Report progress both in the console and in the terminal title
handlers(c("txtprogressbar", "terminal_title"))
with_progress({ y <- slow_sum_p(1:10) })
print(y)
```

handler_tkprogressbar *Progression Handler: Progress Reported as a Tcl/Tk Progress Bars in the GUI*

Description

A progression handler for `tcltk::tkProgressBar()`.

Usage

```
handler_tkprogressbar(
  intrusiveness = getOption("progressr.intrusiveness.gui", 1),
  target = "terminal",
  inputs = list(title = NULL, label = "message"),
  ...
)
```

Arguments

`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`inputs`	(named list) Specifies from what sources the MS Windows progress elements 'title' and 'label' should be updated. Valid sources are `"message"`, `"sticky_message"` and `"non_sticky_message"`, where `"message"` is short for `c("non_sticky_message", "sticky_message")`. For example, `inputs = list(title = "sticky_message", label = "message")` will update the 'title' component from sticky messages only, whereas the 'label' component is updated using any message.
`...`	Additional arguments passed to `tcltk::tkProgressBar()` and `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires the **tcltk** package and that the current R session supports Tcl/Tk (`capabilities("tcltk")`).

Examples

```
if (capabilities("tcltk") && requireNamespace("tcltk", quietly = TRUE)) {

  handlers("tkprogressbar")
  with_progress({ y <- slow_sum_p(1:10) })
  print(y)
```

```
}
```

`handler_tutorial` *A progressr handler useful for Futureverse tutorials*

Description

A progressr handler useful for Futureverse tutorials

Usage

```
handler_tutorial(
  ...,
  spec = c("progress", "cli"),
  type = "time",
  clear = FALSE
)
```

Arguments

`spec`	The progress handler to use.
`type`	Controls which predefined handler format string to use.
`clear`	(logical) If TRUE, any output, typically visual, produced by a reporter will be cleared/removed upon completion, if possible.
`...`	Arguments passed to the selected progress handler.

Details

The default is to use the **progress** package. Contrary to the **cli** package, the **progress** package has the capacity to display *all* progress updates, even when receiving progress updates at a high rate from, for example, parallel processing.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Examples

```
# Display progress and keep the progress output when done
handlers("tutorial")

# Display progress and keep every progress reported
handlers("tutorial", "newline")
```

handler_txtprogressbar *Progression Handler: Progress Reported as Plain Progress Bars (Text) in the Terminal*

Description

A progression handler for `utils::txtProgressBar()`.

Usage

```
handler_txtprogressbar(
  char = "=",
  style = 3L,
  file = stderr(),
  intrusiveness = getOption("progressr.intrusiveness.terminal", 1),
  target = "terminal",
  ...
)
```

Arguments

`char`	(character) The symbols to form the progress bar for `utils::txtProgressBar()`. Contrary to `txtProgressBar()`, this handler supports also ANSI-colored symbols.
`style`	(integer) The progress-bar style according to `utils::txtProgressBar()`.
`file`	(connection) A base::connection to where output should be sent.
`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`...`	Additional arguments passed to `utils::txtProgressBar()` and `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Appearance

Below are a few examples on how to use and customize this progress handler. In all cases, we use `handlers(global = TRUE)`.

```
library(progressr)
handlers("txtprogressbar")
y <- slow_sum_p(1:25)
```

```
library(progressr)
handlers(handler_txtprogressbar(style = 1L))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_txtprogressbar(style = 3L))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_txtprogressbar(char = "#"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_txtprogressbar(char = "<>"))
y <- slow_sum_p(1:25)

library(progressr)
handlers(handler_txtprogressbar(char = cli::col_red(cli::symbol$heart)))
y <- slow_sum_p(1:25)
```

Examples

```
handlers("txtprogressbar")

with_progress({ y <- slow_sum_p(1:10) })
print(y)
```

`handler_void` *Progression Handler: No Progress Report*

Description

Progression Handler: No Progress Report

Usage

```
handler_void(intrusiveness = 0, target = "void", enable = FALSE, ...)
```

Arguments

`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`enable`	(logical) If FALSE, then progress is not reported.
`...`	Additional arguments passed to `make_progression_handler()`.

Details

This progression handler gives not output - it is invisible and silent.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Examples

```
## Not run:
handlers(handler_void())
with_progress(y <- slow_sum_p(1:100))
print(y)

## End(Not run)
```

---

handler_winprogressbar

Progression Handler: Progress Reported as a MS Windows Progress Bars in the GUI

Description

A progression handler for `winProgressBar()` in the **utils** package.

Usage

```
handler_winprogressbar(
  intrusiveness = getOption("progressr.intrusiveness.gui", 1),
  target = "gui",
  inputs = list(title = NULL, label = "message"),
  ...
)
```

Arguments

`intrusiveness`	(numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user.
`target`	(character vector) Specifies where progression updates are rendered.
`inputs`	(named list) Specifies from what sources the MS Windows progress elements 'title' and 'label' should be updated. Valid sources are `"message"`, `"sticky_message"` and `"non_sticky_message"`, where `"message"` is short for `c("non_sticky_message", "sticky_message")`. For example, `inputs = list(title = "sticky_message", label = "message")` will update the 'title' component from sticky messages only, whereas the 'label' component is updated using any message.
`...`	Additional arguments passed to `tcltk::winProgressBar()` and `make_progression_handler()`.

Value

A function of class `progression_handler` that takes a progression condition as its first and only argument.

Requirements

This progression handler requires MS Windows.

Examples

```
## Not run:
handlers(handler_winprogressbar())
with_progress(y <- slow_sum_p(1:100))

## End(Not run)
```

`handlers` *Control How Progress is Reported*

Description

Control How Progress is Reported

Usage

```
handlers(
  ...,
  append = FALSE,
  on_missing = c("error", "warning", "ignore"),
  default = handler_txtprogressbar,
  global = NULL
)
```

Arguments

`append`	(logical) If FALSE, the specified progression handlers replace the current ones, otherwise they are appended to them.
`on_missing`	(character) If `"error"`, an error is thrown if one of the progression handlers does not exist. If `"warning"`, a warning is produced and the missing handler is ignored. If `"ignore"`, the missing handler is ignored.
`default`	The default progression calling handler to use if none are set.
`global`	If TRUE, then the global progression handler is enabled. If FALSE, it is disabled. If NA, then TRUE is returned if it is enabled, otherwise FALSE. Enabling or disabling it requires that there are no condition handlers active, which is a limitation of R's `globalCallingHandlers()`. For example, it cannot be done via `base::tryCatch()`, `base::withCallingHandlers()`, or `base::suppressMessages()`,

or from within a package's `.onLoad()` or `.onAttach()` function. Trying to do so results in an error.

`...` One or more progression handlers, specified as functions, handler names (character strings), or a mix of such. Strings may hold comma-separated handler names.

Details

This function provides a convenient alternative for getting and setting option '`progressr.handlers`'.

Value

(invisibly) the previous list of progression handlers set. If no arguments are specified, then the current set of progression handlers is returned. If `global` is specified, then TRUE is returned if the global progression handler is enabled, otherwise FALSE. If both handlers and `global` are specified, then the previous list of progression handlers is returned.

For package developers

IMPORTANT: Setting progression handlers is a privilege that should be left to the end user. It should not be used by R packages, whose only task is to *signal* progress updates, not to decide if, when, and how progress should be reported.

If you have to set or modify the progression handlers inside a function, please make sure to undo the settings afterward. If not, you will break whatever progression settings the user already has for other purposes used elsewhere. To undo your settings, you can do:

```
old_handlers <- handlers(c("beepr", "progress"))
on.exit(handlers(old_handlers), add = TRUE)
```

It is not possible for a package to enable global progression handling when it is loaded or attached, i.e. by calling `handlers(global = TRUE)` in `.onLoad()` or `.onAttach()`. This is because R always calls those functions via `tryCatch()`, and R does not allow global calling handlers to be registered when there are condition handlers active. Instead, enabling global progression handling is up to the end user, e.g. in their '`~/.Rprofile`' startup file, as explained below.

Configuring progression handling during R startup

A convenient place to configure the default progression handler and to enable global progression reporting by default is in the '`~/.Rprofile`' startup file. For example, the following will (i) cause your interactive R session to use global progression handler by default, and (ii) report progress via the **progress** package when in the terminal and via the RStudio Jobs progress bar when in the RStudio Console. To configure this, add the following to your '`~/.Rprofile`' startup file:

```
if (interactive() && requireNamespace("progressr", quietly = TRUE)) {
  ## Enable global progression updates
  if (getRversion() >= 4) progressr::handlers(global = TRUE)

  ## In RStudio Console, or not?
```

```
  if (Sys.getenv("RSTUDIO") == "1" && !nzchar(Sys.getenv("RSTUDIO_TERM"))) {
    options(progressr.handlers = progressr::handler_rstudio)
  } else {
    options(progressr.handlers = progressr::handler_progress)
  }
}
```

Examples

```
handlers("txtprogressbar")
if (requireNamespace("beepr", quietly = TRUE))
  handlers("beepr", append = TRUE)

with_progress({ y <- slow_sum_p(1:5) })
print(y)


if (getRversion() >= "4.0.0") {

  handlers("txtprogressbar", global = TRUE)
  y <- slow_sum_p(1:4)
  z <- slow_sum_p(6:9)

  handlers(global = FALSE)
}
```

`progress_progressr` *Use Progressr with Plyr Map-Reduce Functions*

Description

A "progress bar" for **plyr**'s `.progress` argument.

Usage

```
progress_progressr(...)
```

Arguments

`...`	Not used.

Value

A named base::list that can be passed as argument `.progress` to any of **plyr** function accepting that argument.

Limitations

One can use use doFuture::registerDoFuture() to run **plyr** functions in parallel, e.g. `plyr::l_ply(..., .parallel = TRUE)`. Unfortunately, using `.parallel = TRUE` disables progress updates because, internally, **plyr** forces `.progress = "none"` whenever `.parallel = TRUE`. Thus, despite the **future** ecosystem and **progressr** would support it, it is not possible to run **dplyr** in parallel *and* get progress updates at the same time.

Examples

```
if (requireNamespace("plyr", quietly=TRUE)) {

  with_progress({
    y <- plyr::llply(1:10, function(x) {
      Sys.sleep(0.1)
      sqrt(x)
    }, .progress = "progressr")
  })

}
```

---

progressor *Create a Progressor Function that Signals Progress Updates*

Description

Create a Progressor Function that Signals Progress Updates

Usage

```
progressor(
  steps = length(along),
  along = NULL,
  offset = 0L,
  scale = 1L,
  transform = function(steps) scale * steps + offset,
  message = character(0L),
  label = NA_character_,
  trace = FALSE,
  initiate = TRUE,
  finalize = FALSE,
  auto_finish = TRUE,
  on_exit = !identical(envir, globalenv()),
  enable = getOption("progressr.enable", TRUE),
  envir = parent.frame()
)
```

Arguments

`steps`	(integer) Number of progressing steps.
`along`	(vector; alternative) Alternative that sets `steps = length(along)`.
`offset, scale`	(integer; optional) scale and offset applying the transform `steps <- scale * steps + offset`.
`transform`	(function; optional) A function that takes the effective number of `steps` as input and returns another finite and non-negative number of steps.
`message`	(character vector or a function) If a character vector, then it is pasted together into a single string using an empty separator. If a function, then the message is constructed by `conditionMessage(p)` calling this function with the progression condition `p` itself as the first argument.
`label`	(character) A label.
`trace`	(logical) If TRUE, then the call stack is recorded, otherwise not.
`initiate`	(logical) If TRUE, the progressor will signal a progression 'initiate' condition when created.
`finalize`	(logical) If TRUE, the progressor signals a progression 'shutdown' condition when finalized by the garbage collector.
`auto_finish`	(logical) If TRUE, then the progressor will signal a progression 'finish' condition as soon as the last step has been reached.
`on_exit, envir`	(logical) If TRUE, then the created progressor will signal a progression 'finish' condition when the calling frame exits. This is ignored if the calling frame (`envir`) is the global environment.
`enable`	(logical) If TRUE, progression conditions are signaled when calling the progressor function created by this function. If FALSE, no progression conditions is signaled because the progressor function is an empty function that does nothing.

Details

A `progressor` function can only be created inside a local environment, e.g. inside a function, within a `local()` call, or within a `with_progress()` call. Notably, it *cannot* be created at the top level, e.g. immediately at the R prompt or outside a local environment in an R script. If attempted, an informative error message is produced, e.g.

```
> p <- progressr::progressor(100)
Error in progressr::progressor(100) :
  A progressor must not be created in the global environment unless
wrapped in a with_progress() or without_progress() call. Alternatively,
create it inside a function or in a local() environment to make sure
there is a finite life span of the progressor
```

Value

A function of class `progressor`.

progressr *progressr: A Unifying API for Progress Updates*

Description

The **progressr** package provides a minimal, unifying API for scripts and packages to report progress updates from anywhere including when using parallel processing.

Details

The package is designed such that *the developer* can to focus on *what* progress should be reported on without having to worry about *how* to present it.

The *end user* has full control of *how*, *where*, and *when* to render these progress updates. For instance, they can chose to report progress in the terminal using `utils::txtProgressBar()` (default) or `cli::cli_progress_bar()`, via the R graphical user interface (GUI) using `utils::winProgressBar()` or `tcltk::tkProgressBar()`, or via the RStudio GUI using `rstudioapi::jobSetProgress()`. An alternative to above visual rendering, is to report progress as audio using `beepr::beep()` or `BRRR::skrrrahh()`. It is also possible to use a combination of above progression handlers, e.g. a progress bar in the terminal together with audio updates. Besides the existing handlers, it is possible to develop custom progression handlers.

The **progressr** package is compatible with **shiny** applications and **knitr** rendering.

The **progressr** package uses R's condition framework for signaling progress updated. Because of this, progress can be reported from almost anywhere in R, e.g. from classical for and while loops, from map-reduce APIs like the `lapply()` family of functions, **purrr**, **plyr**, and **foreach**. The **progressr** package will also work with parallel processing via the **future** framework, e.g. `future.apply::future_lapply()`, `furrr::future_map()`, and `foreach::foreach()` with **doFuture**.

Progression Handlers

In the terminal:

- handler_txtprogressbar (built-in; default)
- handler_pbcol (built-in)
- handler_ascii_alert (built-in)
- handler_pbmcapply (requires the **pbmcapply** package)
- handler_progress (requires the **progress** package)
- handler_terminal_title (built-in; in the terminal title)

In a graphical user interface (GUI):

- handler_rstudio (only in the RStudio Console)
- handler_tkprogressbar (built-in)
- handler_winprogressbar (built-in)

As sound:

- handler_ascii_alert (built-in)
- handler_beepr (requires the **beepr** package)
- handler_brrr (requires the **BRRR** package)

Via the file system:

- handler_filesize (built-in)

In Shiny:

- withProgressShiny (requires the **shiny** package)

Via notification systems:

- handler_ntfy (requires the **ntfy** package)
- handler_notifier (requires the non-CRAN **notifier** package)
- handler_rpushbullet (requires the **RPushBullet** package)

Miscellaneous:

- handler_debug (built-in)
- handler_newline (built-in)
- handler_slowdown (built-in)
- handler_void (built-in)

Author(s)

Maintainer: Henrik Bengtsson <henrikb@braju.com> (ORCID) [copyright holder]

Authors:

- Henrik Bengtsson <henrikb@braju.com> (ORCID) [copyright holder]

See Also

Useful links:

- https://progressr.futureverse.org
- https://github.com/futureverse/progressr
- Report bugs at https://github.com/futureverse/progressr/issues

Examples

```
library(progressr)

xs <- 1:5

with_progress({
  p <- progressor(along = xs)
  y <- lapply(xs, function(x) {
    Sys.sleep(0.1)
    p(sprintf("x=%g", x))
    sqrt(x)
  })
})
```

`progressr.options` *Options and environment variables used by the 'progressr' packages*

Description

Below are environment variables and R options that are used by the **progressr** package. Below are all R options that are currently used by the **progressr** package.

WARNING: Note that the names and the default values of these options may change in future versions of the package. Please use with care until further notice.

Options for controlling progression reporting

'`progressr.handlers`': (function, list of functions, or one or more handler names) Zero or more progression handlers that will report on any progression updates. The R option value is parsed, expanded, and validated by `handlers()`. If an empty list, progress updates are ignored. If NULL, the default (`handler_txtprogressbar`) progression handlers is used. The recommended way to set this option is via `handlers()`. (Default: NULL)

Options for controlling progression handlers

'`progressr.clear`': (logical) If TRUE, any output, typically visual, produced by a reporter will be cleared/removed upon completion, if possible. (Default: TRUE)

'`progressr.enable`': (logical) If FALSE, then progress is not reported. (Default: TRUE in interactive mode, otherwise FALSE)

This option must be set *before* setting up calling handlers via `handler_nnn()` functions, e.g. `handlers(handler_txtprogressbar())`. This is not needed when using `handlers("txtprogressbar")`.

'`progressr.enable_after`': (numeric) Delay (in seconds) before progression updates are reported. (Default: `0.0`)

'`progressr.times`': (numeric) The maximum number of times a handler should report progression updates. If zero, then progress is not reported. (Default: `+Inf`)

'progressr.interval': (numeric) The minimum time (in seconds) between successive progression updates from this handler. (Default: 0.0)

'progressr.intrusiveness': (numeric) A non-negative scalar on how intrusive (disruptive) the reporter to the user. This multiplicative scalar applies to the *interval* and *times* parameters. (Default: 1.0)

'progressr.intrusiveness.audio': (numeric) intrusiveness for auditory progress handlers (Default: 5.0)

'progressr.intrusiveness.file': (numeric) intrusiveness for file-based progress handlers (Default: 5.0)

'progressr.intrusiveness.gui': (numeric) intrusiveness for graphical-user-interface progress handlers (Default: 1.0)

'progressr.intrusiveness.notifier': (numeric) intrusiveness for progress handlers that creates notifications (Default: 10.0)

'progressr.intrusiveness.terminal': (numeric) intrusiveness for progress handlers that outputs to the terminal (Default: 1.0)

'progressr.intrusiveness.debug': (numeric) intrusiveness for "debug" progress handlers (Default: 0.0)

Options for controlling how standard output and conditions are relayed

'progressr.delay_conditions': (character vector) condition classes to be captured and relayed at the end after any captured standard output is relayed. (Default: c("condition"))

'progressr.delay_stdout': (logical) If TRUE, standard output is captured and relayed at the end just before any captured conditions are relayed. (Default: TRUE)

Options for controlling interrupts

'progressr.interrupts': (logical) Controls whether interrupts should be detected or not. If FALSE, then interrupts are not detected and progress information is generated. (Default: TRUE)

'progressr.delay_stdout': (logical) If TRUE, standard output is captured and relayed at the end just before any captured conditions are relayed. (Default: TRUE)

Options for debugging progression updates

'progressr.debug': (logical) If TRUE, extensive debug messages are generated. (Default: FALSE)

Options for progressr examples and demos

'progressr.demo.delay': (numeric) Delay (in seconds) between each iteration of slow_sum_p() and slow_sqrt_p(). (Default: 1.0 and 0.5)

'progressr.demo.stdout': (logical) If TRUE, then a text is outputted to the standard output per element. (Default: FALSE)

'progressr.demo.message': (logical) If TRUE, then a message is outputted per element. (Default: TRUE)

'progressr.demo.sticky': (logical) If TRUE, then a "sticky" message is outputted every ten element. (Default: TRUE)

'progressr.demo.progress': (logical) If TRUE, then a progressor is created, otherwise a void progressor that does nothing is used. (Default: TRUE)

Environment variables that set R options

Some of the above R 'progressr.*' options can be set by corresponding environment variable R_PROGRESSR_* *when the* **progressr** *package is loaded*. For example, if R_PROGRESSR_ENABLE = "true", then option 'progressr.enable' is set to TRUE (logical). For example, if R_PROGRESSR_ENABLE_AFTER = "2.0", then option 'progressr.enable_after' is set to 2.0 (numeric). Environment variable R_PROGRESSR_HANDLERS sets option 'progressr.handlers', e.g. R_PROGRESSR_HANDLERS="cli,beepr".

See Also

To set R options when R starts (even before the **progressr** package is loaded), see the Startup help page. The **startup** package provides a friendly mechanism for configuring R at startup.

---

with_progress *Report on Progress while Evaluating an R Expression*

Description

Report on Progress while Evaluating an R Expression

Usage

```
with_progress(
  expr,
  handlers = progressr::handlers(),
  cleanup = TRUE,
  delay_terminal = NULL,
  delay_stdout = NULL,
  delay_conditions = NULL,
  interrupts = getOption("progressr.interrupts", TRUE),
  interval = NULL,
  enable = NULL
)

without_progress(expr)
```

Arguments

expr	An R expression to evaluate.
handlers	A progression handler or a list of them. If NULL or an empty list, progress updates are ignored.

`cleanup` If TRUE, all progression handlers will be shutdown at the end regardless of whether the progression is complete or not.

`delay_terminal` If TRUE, output and conditions that may end up in the terminal will be delayed.

`delay_stdout` If TRUE, standard output is captured and relayed at the end just before any captured conditions are relayed.

`delay_conditions` A character vector specifying base::condition classes to be captured and relayed at the end after any captured standard output is relayed.

`interrupts` Controls whether interrupts should be detected or not. If TRUE and an interrupt is signaled, progress handlers are asked to report on the current amount of progress when the evaluation was terminated by the interrupt, e.g. when a user pressed Ctrl-C in an interactive session, or a batch process was interrupted because it ran out of time. Note that it's optional for a progress handler to support this and only some do.

`interval` (numeric) The minimum time (in seconds) between successive progression updates from handlers.

`enable` (logical) If FALSE, then progress is not reported. The default is to report progress in interactive mode but not batch mode. See below for more details.

Details

If you are writing a Shiny app, use the `withProgressShiny()` function instead of this one.

If the global progression handler is enabled, it is temporarily disabled while evaluating the `expr` expression.

If a `progression` condition causes the progressor to be completed ("reaches 100%"), then `with_progress()` will muffle the `progression` condition preventing it from reaching, say, any global condition handlers.

IMPORTANT: This function is meant for end users only. It should not be used by R packages, whose only task is to *signal* progress updates, not to decide if, when, and how progress should be reported.

`without_progress()` evaluates an expression while ignoring all progress updates.

Value

Returns the value of the expression.

Progression handler functions

Formally, progression handlers are calling handlers that are called when a progression condition is signaled. These handlers are functions that take one argument which is the progression condition.

Progress updates in batch mode

When running R from the command line, R runs in a non-interactive mode (`interactive()` returns `FALSE`). The default behavior of `with_progress()` is to *not* report on progress in non-interactive

mode. To have progress being reported also then, set R options 'progressr.enable' or environment variable R_PROGRESSR_ENABLE to TRUE. Alternatively, one can set argument enable=TRUE when calling with_progress(). For example,

```
$ Rscript -e "library(progressr)" -e "with_progress(slow_sum_p(1:5))"
```

will *not* report on progress, whereas:

```
$ export R_PROGRESSR_ENABLE=TRUE
$ Rscript -e "library(progressr)" -e "with_progress(slow_sum_p(1:5))"
```

will.

See Also

For Shiny apps, use withProgressShiny() instead of this function. Internally, this function is built around base::withCallingHandlers().

Examples

```
## The slow_sum_p() example function
slow_sum_p <- progressr::slow_sum_p
print(slow_sum_p)

x <- 1:10

## Without progress updates
y <- slow_sum_p(x)


## Progress reported via txtProgressBar (default)
handlers("txtprogressbar")  ## default
with_progress({
  y <- slow_sum_p(x)
})

## Progress reported via tcltk::tkProgressBar
if (capabilities("tcltk") && requireNamespace("tcltk", quietly = TRUE)) {
  handlers("tkprogressbar")
  with_progress({
    y <- slow_sum_p(x)
  })
}

## Progress reported via progress::progress_bar)
if (requireNamespace("progress", quietly = TRUE)) {
  handlers("progress")
  with_progress({
    y <- slow_sum_p(x)
  })
}
```

```
## Progress reported via txtProgressBar and beepr::beep
if (requireNamespace("beepr", quietly = TRUE)) {
  handlers("beepr", "txtprogressbar")
  with_progress({
    y <- slow_sum_p(x)
  })
}

## Progress reported via customized utils::txtProgressBar and beepr::beep,
## if available.
handlers(handler_txtprogressbar(style = 3L))
if (requireNamespace("beepr", quietly = TRUE)) {
  handlers("beepr", append = TRUE)
}

with_progress({
  y <- slow_sum_p(1:30)
})
```

`withProgressShiny` *Use Progressr in Shiny Apps: Plug-in Backward-Compatible Replacement for shiny::withProgress()*

Description

A plug-in, backward-compatible replacement for `shiny::withProgress()`.

Usage

```
withProgressShiny(
  expr,
  ...,
  message = NULL,
  detail = NULL,
  inputs = list(message = NULL, detail = "message"),
  env = parent.frame(),
  quoted = FALSE,
  handlers = c(shiny = handler_shiny, progressr::handlers(default = NULL))
)
```

Arguments

`expr`, `...`, `env`, `quoted`
: Arguments passed to `shiny::withProgress()` as is.

`message`, `detail`
: (character string) The message and the detail message to be passed to `shiny::withProgress()`.

inputs	(named list) Specifies from what sources the Shiny progress elements 'message' and 'detail' should be updated. Valid sources are `"message"`, `"sticky_message"` and `"non_sticky_message"`, where `"message"` is short for `c("non_sticky_message", "sticky_message")`. For example, `inputs = list(message = "sticky_message", detail = "message")` will update the Shiny 'message' component from sticky messages only, whereas the 'detail' component is updated using any message.
handlers	Zero or more progression handlers used to report on progress.

Value

The value of shiny::withProgress.

Requirements

This function requires the **shiny** package and will use the `handler_shiny()` **progressr** handler internally to report on updates.

Examples

```
if (requireNamespace("shiny", quietly = TRUE)) {

  library(shiny)
  library(progressr)

  app <- shinyApp(
    ui = fluidPage(
      plotOutput("plot")
    ),

    server = function(input, output) {
      output$plot <- renderPlot({
        X <- 1:15
        withProgressShiny(message = "Calculation in progress",
                          detail = "Starting ...",
                          value = 0, {
          p <- progressor(along = X)
          y <- lapply(X, FUN=function(x) {
            Sys.sleep(0.25)
            p(sprintf("x=%d", x))
          })
        })

        plot(cars)

        ## Terminate the Shiny app
        Sys.sleep(1.0)
        stopApp(returnValue = invisible())
      })
    }
  )

  local({
```

```
    oopts <- options(device.ask.default = FALSE)
    on.exit(options(oopts))
    if (interactive()) print(app)
  })

}
```

Index

∗ **iteration**
progressr, 33
∗ **programming**
progressr, 33

base::condition, *38*
base::connection, *3*, *15*, *22*, *25*
base::list, *30*
base::suppressMessages(), *28*
base::tryCatch(), *28*
base::withCallingHandlers(), *28*, *39*
beepr::beep(), *4*, *33*

cli::cli_progress_bar(), *7, 8*, *33*

doFuture::registerDoFuture(), *31*

foreach::foreach(), *33*
furrr::future_map(), *33*
future.apply::future_lapply(), *33*

handler_ascii_alert, 3, *33, 34*
handler_beepr, 4, *34*
handler_brrr, 5, *34*
handler_callback, 6
handler_cli, 7
handler_debug, 9, *34*
handler_filesize, 10, *34*
handler_newline, *34*
handler_notifier, 11, *34*
handler_ntfy, 12, *34*
handler_pbcol, 13, *33*
handler_pbmcapply, 14, *33*
handler_progress, 16, *33*
handler_rpushbullet, 18, *34*
handler_rstudio, 19, *33*
handler_shiny(), *41*
handler_slowdown, 20, *34*
handler_terminal_title, 21, *33*
handler_tkprogressbar, 23, *33*
handler_tutorial, 24
handler_txtprogressbar, 25, *33*
handler_txtprogressbar(), *15*
handler_void, 26, *34*
handler_winprogressbar, 27, *33*
handlers, 28
handlers(), *35*

lapply(), *33*

make_progression_handler(), *3–6*, *8–12*, *14–16*, *18, 19*, *21–23*, *25–27*

ntfy::ntfy_send(), *12*

pbmcapply::progressBar(), *14, 15*
Progress bar variables, *8*
progress::progress_bar, *16*
progress::progress_bar(), *16*
progress_progressr, 30
progression, *3–6*, *8–12*, *14–16*, *18, 19*, *21–25*, *27, 28*, *32*, *38*
progressor, 31
progressr, 33
progressr-package *(*progressr*)*, 33
progressr.clear *(*progressr.options*)*, 35
progressr.debug *(*progressr.options*)*, 35
progressr.delay_conditions *(*progressr.options*)*, 35
progressr.delay_stdout *(*progressr.options*)*, 35
progressr.demo.delay *(*progressr.options*)*, 35
progressr.demo.message *(*progressr.options*)*, 35
progressr.demo.progress *(*progressr.options*)*, 35
progressr.demo.stdout *(*progressr.options*)*, 35
progressr.demo.sticky *(*progressr.options*)*, 35

progressr.enable *(progressr.options)*, 35
progressr.enable_after
(progressr.options), 35
progressr.handlers *(progressr.options)*,
35
progressr.interrupts
(progressr.options), 35
progressr.interval *(progressr.options)*,
35
progressr.intrusiveness
(progressr.options), 35
progressr.options, 35
progressr.times *(progressr.options)*, 35

RPushbullet::pbPost(), *18*

shiny::withProgress, *41*
shiny::withProgress(), *40*
slow_sqrt_p(), *36*
slow_sum_p(), *36*
Startup, *37*

tcltk::tkProgressBar(), *23*, *33*

utils::txtProgressBar(), *15*, *25*, *33*

with_progress, 37
without_progress *(with_progress)*, 37
withProgressShiny, *34*, 40
withProgressShiny(), *38, 39*