Package: progressr (via r-universe)

October 30, 2024

Version 0.15.0

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. future.apply::future_lapply(), furrr::future_map(), and 'foreach' with 'doFuture'. The package is compatible with Shiny applications.

License GPL (>= 3)

Depends R (>= 3.5.0)

Imports digest, utils

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

VignetteBuilder progressr

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

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

RoxygenNote 7.3.2

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph]
(<https://orcid.org/0000-0002-7579-5165>)

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2024-10-29 05:40:02 UTC

Contents

`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

`...`	One or more progression handlers. Alternatively, this functions accepts also a single vector of progression handlers as input. If this vector is empty, then an empty set of progression handlers will be set.
`append`	(logical) If FALSE, the specified progression handlers replace the current ones, otherwise appended to them.
`on_missing`	(character) If `"error"`, an error is thrown if one of the progression handlers does not exists. If `"warning"`, a warning is produces and the missing handlers is ignored. If `"ignore"`, the missing handlers 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. Argument `global` must not used with other arguments.

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 handlers is enabled, otherwise false.

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, which 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 you settings, you can do:

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

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. handler_txtprogressbar, other whenever using the RStudio Console, 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(1:5) })
print(y)


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

  handlers(global = TRUE)
  y <- slow_sum(1:4)
  z <- slow_sum(6:9)

  handlers(global = FALSE)
}
```

---

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()`.

Examples

```
handlers("ascii_alert")
with_progress({ y <- slow_sum(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()`.

Requirements

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

Examples

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

  handlers("beepr")
  with_progress({ y <- slow_sum(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",
  ...
)
```

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.
`...`	Additional arguments passed to `make_progression_handler()`.

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)`.

```
handlers("cli")
y <- slow_sum(1:25)

handlers(handler_cli(format = "{cli::pb_spin} {cli::pb_bar} {cli::pb_current}/{cli::pb_total} {cli::p
y <- slow_sum(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(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().

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(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.

Examples

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

## End(Not run)
```

---

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()`.

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)`.

```
handlers("pbcol")
y <- slow_sum(1:25)

handlers(handler_pbcol(adjust = 0.5))
y <- slow_sum(1:25)

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(1:25)
```

Examples

```
handlers(handler_pbcol)
with_progress({ y <- slow_sum(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 `make_progression_handler()`.

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.

```
handlers("pbmcapply")
y <- slow_sum(1:25)
```

Examples

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

  handlers("pbmcapply")
  with_progress({ y <- slow_sum(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 = ":spin [:bar] :percent :message",
  show_after = 0,
  intrusiveness = getOption("progressr.intrusiveness.terminal", 1),
  target = "terminal",
  ...
)
```

Arguments

`format`	(character string) The format of the progress bar.
`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.
`...`	Additional arguments passed to `make_progression_handler()`.

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)`.

```
handlers("progress")
y <- slow_sum(1:25)

handlers(handler_progress(complete = "#"))
y <- slow_sum(1:25)

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

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

Examples

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

  handlers(handler_progress(format = ":spin [:bar] :percent :message"))
  with_progress({ y <- slow_sum(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().

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(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 `make_progression_handler()`.

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(1:10) })
  print(y)

}
```

`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 `make_progression_handler()`.

Appearance

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

```
handlers("txtprogressbar")
y <- slow_sum(1:25)

handlers(handler_txtprogressbar(style = 1L))
y <- slow_sum(1:25)

handlers(handler_txtprogressbar(style = 3L))
y <- slow_sum(1:25)

handlers(handler_txtprogressbar(char = "#"))
y <- slow_sum(1:25)

handlers(handler_txtprogressbar(char = "<>"))
y <- slow_sum(1:25)

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

Examples

```
handlers("txtprogressbar")

with_progress({ y <- slow_sum(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.

Examples

```
## Not run:
handlers(handler_void())
with_progress(y <- slow_sum(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 `make_progression_handler()`.

Requirements

This progression handler requires MS Windows.

Examples

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

## End(Not run)
```

---

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,
  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 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.
`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 create 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()` or `progress::progress_bar()` or via the graphical user interface (GUI) using `utils::winProgressBar()`

or `tcltk::tkProgressBar()`. An alternative to above visual rendering of progress, is to report it using `beepr::beep()` sounds. It is 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 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**.

The **progressr** package is compatible with Shiny applications.

Progression Handlers

In the terminal:

- handler_txtprogressbar (default)
- handler_pbcol
- handler_pbmcapply
- handler_progress
- handler_ascii_alert
- handler_debug

In a graphical user interface (GUI):

- handler_rstudio
- handler_tkprogressbar
- handler_winprogressbar

As sound:

- handler_beepr
- handler_ascii_alert

Via the file system:

- handler_filesize

In Shiny:

- withProgressShiny

Via notification systems:

- handler_ntfy
- handler_notifier
- handler_rpushbullet

Author(s)

Maintainer: 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 or list of functions) Zero or more progression handlers that will report on any progression updates. If 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)

'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()`. (Default: `1.0`)

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).

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.

`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")
  })

}
```

`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

```
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)
})
```

`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 the progression is complete or not.
`delay_terminal`	If TRUE, output and conditions that may end up in the terminal will 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 a interrupt is signaled, progress handlers are asked to report on the current amount 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.

IMPORTANT: This function is meant for end users only. It should not be used by R packages, which 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 takes 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 on 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(1:5))"
```

will *not* report on progress, whereas:

```
$ export R_PROGRESSR_ENABLE=TRUE
$ Rscript -e "library(progressr)" -e "with_progress(slow_sum(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() example function
slow_sum <- progressr::slow_sum
print(slow_sum)

x <- 1:10

## Without progress updates
y <- slow_sum(x)
```

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

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

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

## Progress reported via txtProgressBar and beepr::beep
if (requireNamespace("beepr", quietly = TRUE)) {
  handlers("beepr", "txtprogressbar")
  with_progress({
    y <- slow_sum(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(1:30)
})
```

Index