Package 'progressr'

May 19, 2020

Version 0.6.0

Title A 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() or progress::progress_bar(), in a graphical user interface using utils::winProgressBar(), tcltk::tkProgressBar() or shiny::withProgress(), via the speakers using beep::beepr(), 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 APIs 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)

Imports digest, utils

Suggests graphics, tcltk, beepr, pbmcapply, progress, purrr, foreach, plyr, doFuture, future (>= 1.16.0), future.apply, furrr, shiny, commonmark, base64enc, tools

VignetteBuilder progressr

URL https://github.com/HenrikBengtsson/progressr

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

RoxygenNote 7.1.0

NeedsCompilation no

Author Henrik Bengtsson [aut, cre, cph]

Maintainer Henrik Bengtsson <henrikb@braju.com>

Repository CRAN

Date/Publication 2020-05-19 06:30:02 UTC

R topics documented:

`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
)
```

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.

Details

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

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.*

Value

(invisibly) the previous list of progression handlers set. If no arguments are specified, then the current set of progression handlers is returned.

Examples

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

with_progress({ y <- slow_sum(10) })
print(y)
```

---

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.auditory", 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,
  intrusiveness = getOption("progressr.intrusiveness.auditory", 5),
  target = "audio",
  ...
)
```

Arguments

`initiate, update, finish`	(integer) Indices of `beepr::beep()` sounds to play when progress starts, is updated, and completes. 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_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",
  ...
)
```

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.
`...`	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",
  ...
)
```

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

Description

A progression handler for `pbmcapply::progressBar()`.

Usage

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

Arguments

`substyle`	(integer) The progress-bar substyle according to `pbmcapply::progressBar()`.
`style`	(character) The progress-bar style according to `pbmcapply::progressBar()`.
`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

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

With `handlers(handler_pbmcapply())`:

```
|                                          |   0%, ETA NA
|===========                               |  30%, ETA 01:32
|======================================== |  99%, ETA 00:01
```

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

With `handlers(handler_progress())`:

```
[-------------------------------------------------]   0%
[====>--------------------------------------------]  10%
[================================================>]  99%
```

If the progression updates have messages, they will appear like:

```
[----------------------------------------]   0% Starting
[===========>----------------------------]  30% Importing
[=====================================>]  99% Summarizing
```

Examples

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

  handlers(handler_progress(format = ":spin [:bar] :percent :message"))
  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",
  ...
)
```

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()`.

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

Arguments

`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 is how this progress handler renders at 0%, 30% and 99% progress for the three different `style` values that `utils::txtProgressBar()` supports.

With `handlers(handler_txtprogressbar(style = 1L))`:

```
==================================
============================================================
```

With handlers(handler_txtprogressbar(style = 2L)):

```
==================================
============================================================
```

With handlers(handler_txtprogressbar(style = 3L)):

```
  |                                                      |   0%
  |===============                                       |  30%
  |==================================================|  99%
```

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",
  ...
)
```

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()`.

Requirements

This progression handler requires MS Windows.

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,
  label = NA_character_,
  initiate = TRUE,
  auto_finish = TRUE
)
```

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.
`label`	(character) A label.
`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.

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_pbmcapply
- handler_progress
- handler_ascii_alert
- handler_debug

In the graphical user interface (GUI):

- handler_tkprogressbar
- handler_winprogressbar

Via audio:

- handler_beepr
- handler_ascii_alert

Via the file system:

- handler_filesize

In Shiny:

- withProgressShiny

Examples

```
library(progressr)

xs <- 1:5

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

`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 Compatibility Replacement for shiny::withProgress()*

Description

Use Progressr in Shiny Apps: Plug-in Backward Compatibility Replacement for shiny::withProgress()

Usage

```
withProgressShiny(
  expr,
  ...,
  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.
`handlers`	Zero or more progression handlers used to report on progress.

Value

The value of shiny::withProgress.

Requirements

This function requires the **shiny** package.

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 = "This may take a while ...", value = 0, {
        p <- progressor(along = X)
        y <- lapply(X, FUN=function(x) {
```

```
            p()
            Sys.sleep(0.25)
          })
        })

        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,
  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.
`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

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

Return nothing (reserved for future usage).

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

`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