Using `lumberjack`

Mark van der Loo

September 12, 2024 | Package version 1.3.1.1

Contents

1 Purpose of this package: tracking changes in data 2

2 Tracking changes in scripts that process data 2

3 A little background 3

4 Controlling where the logging data is written 4

5 Tracking multiple datasets 4

6 Tracking changes in data in interactive sessions 4

7 Available loggers 5

7.1 In the lumberjack package 5

7.2 In other packages 6

8 Properties of the lumberjack pipe operator 6

9 Extending lumberjack 6

9.1 The lumberjack logging API 6

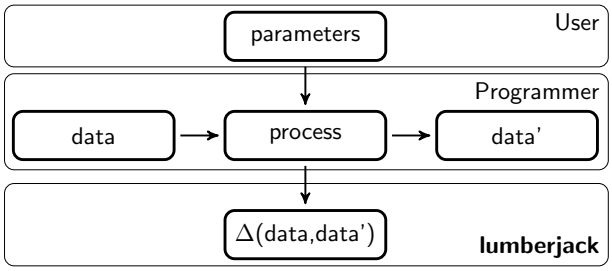
9.2 R6 classes 7

9.3 Reference classes 8

9.4 Advice for package authors 8

1 Purpose of this package: tracking changes in data

This package allows one to monitor changes in data as they get processed, with very little effort. It offers a clear and sharp separation of concerns between the primary goal of a data processing script, and a secondary goal: namely gathering data about the data process itself. The following diagram demonstrates the idea.



A programmer writes a script that transforms `data` into `data'`, possibly based on externally provided parameters. The **lumberjack** package automatically gathers information on how a each process step changed the data. The way the difference between two versions of a data set is computed (Δ, in the diagram) is fully customizable.

2 Tracking changes in scripts that process data

Consider as an example the following simple data analysis script in a file called `process.R`.

```
data(women)
women$height <- women$height * 0.0254
women$weight <- women$weight * 0.453592
women$bmi <- women$weight/(women$height^2)
outfile <- tempfile(fileext=".csv")
write.csv(women, file=outfile, row.names=FALSE)
```

This script loads the `women` dataset, converts height and weight to SI units (meters and kg), and adds a Body-Mass Index column. We can run the code in this file using `source` and read the result from the temporary file in `outfile`.

To check what happens with the `women` dataset at each step we need to do two things. First, we define which dataset must me tracked, in what way, and for what part of the script. This can be done by adding one line of code.

```
data(women)

start_log(women, logger=simple$new())

women$height <- women$height * 0.0254
women$weight <- women$weight * 0.453592
women$bmi <- women$weight/(women$height^2)
outfile <- tempfile(fileext=".csv")
write.csv(women, file=outfile, row.names=FALSE)
```

Second, we run our script using `lumberjack::run_file`.

```
library(lumberjack)
out <- run_file("process.R")
```

All variables created during `run_file` are stored in `out`.

```
head(out$women, 3)
```

```
  height   weight      bmi
1 1.4732 52.16308 24.03476
2 1.4986 53.07026 23.63087
3 1.5240 54.43104 23.43563
```

The logging information is by default written to a file with a name that is the combination of the data set name and the logger name, here: `women_simple.csv` (but this can be customized, see `?simple`).

```
  read.csv("women_simple.csv")
```

```
  step                       time          srcref
1    1 2024-09-12 06:06:45.649352   process.R#5-5
2    2 2024-09-12 06:06:45.650162   process.R#8-8
3    3  2024-09-12 06:06:45.65052 process.R#10-10
4    4 2024-09-12 06:06:45.650862 process.R#12-12
5    5 2024-09-12 06:06:45.651228 process.R#14-14
6    6 2024-09-12 06:06:45.652276 process.R#15-15
                                        expression changed
1 start_log(women, logger = simple$new(verbose=TRUE))   FALSE
2               women$height <- women$height * 0.0254    TRUE
3             women$weight <- women$weight * 0.453592    TRUE
4          women$bmi <- women$weight/(women$height^2)    TRUE
5                outfile <- tempfile(fileext=".csv")   FALSE
6     write.csv(women, file=outfile, row.names=FALSE)   FALSE
```

The `simple` logger records for each expression in the script whether it changed the data that is being tracked. An overview of available loggers is given in Section 7.

Summarizing, to track chages in a data set one needs to do the following.

1. Define a *logger*. Here this is done with `simple$new()`.
2. Tell **lumberjack** which dataset(s) to log. Here, this is done with `start_log(dataset, logger)`. When tracking multiple datasets, each dataset must get its own logger (see Section 5).
3. Develop the analyses as usual.
4. *Optionally* dump the logging information and close the logger explicitly (see Section 4).
5. Run the whole file using `lumberjack::run_file`.

3 A little background

We give a rough sketch on how **lumberjack** works. Two concepts govern its behavior. The first concept is the *logger*. A logger is an R object that capable of comparing two datasets. Depending on the type of logger it can compare various things. The built-in `simple` logger just records whether two versions of a dataset are identical or not. The `cellwise` logger compares two versions of a data set cell-by-cell and records the old and new values if they differ. The `expression_logger` tracks the value of an R expression as the data gets processed. A logger also offers functionality to dump logging information to file or elsewhere.

The second concept is the *runtime*. **lumberjack** intercepts the R expressions written by the user and calls the logger to compare the current version of a data set with the previous version. The runtime also takes care of keeping an old version of the data in memory for comparison. When the user calls `dump_log()` it makes sure that the 'dump' functionality of the active logger(s) is called.

We have already seen the `run_file()` implementation of the **lumberjack** runtime. There is a second implementation for interactive use. This is the so-called lumberjack 'pipe' operator `%L>%`, which is discussed in Section 6.

4 Controlling where the logging data is written

The location of output is determined by the logger when `dump_log` is called. This is done by default by `run_file` after executing the script. All loggers in **lumberjack** write output to a `csv` file with default location `dataset_logger.csv`. If `run_file` is used to to execute an R file, then the log is written in the same directory as where the R file resides. You can control where and when logging information is dumped by calling `dump_log` explicitly.

For the **lumberjack** loggers, `dump_log` has an argument `file` to control explicitly where logging data is saved. For example, to dump logging information for `mtcars` in `hihi.csv` one can do the following.

```
start_log(mtcars, logger=simple$new())
# all data transformations here...
dump_log(file="hihi.csv")
```

Note that we took care to state 'loggers in **lumberjack**' every time. This is because **lumberjack** is extensible and other loggers can be developed that output logs to a data base for example. In fact, the parameters that `dump_log()` accepts, apart from `data`, `logger` and `stop`, can be different for each logger in principle. For a specification of arguments and values, see the help pages for each logger.

5 Tracking multiple datasets

Call `start_log`, with a new logger on each dataset to track. For example to track both `women` and `mtcars` with the `simple` logger, do the following.

```
start_log(women, logger=simple$new())
start_log(mtcars, logger=simple$new())
# all data transformations here...
dump_log()
```

Calling `dump_log()` will cause all loggers to stop tracking changes and write changes to file. To dump all loggers for a specific dataset, provide the dataset when dumping.

```
dump_log(data=mtcars)
```

It is also possible to use multiple loggers on a single dataset. To is is done by calling `start_log` multiple times for the same data set, with different loggers. Here we track the women dataset with the `simple` logger and with the `cellwise` logger.

```
women$id <- 1:15
start_log(women, logger=simple$new())
start_log(women, logger=cellwise(key="id"))
# all data transformations here...
dump_log()
```

Here, the `cellwise` logger records every change in every cell as the data gets processed. It needs a key column to be able to identify and store the location of each cell for each record. To dump a specific logger for a specific dataset, pass the data and the name of the logger.

```
dump_log(women, "cellwise")
```

6 Tracking changes in data in interactive sessions

The **lumberjack** operator is a forward 'pipe' operator that enables logging. In this example we compute again the BMI index of records in the `women` dataset that comes with R. We use the `transform` function from base R to derive the new columns.

```
  data(women)
  women$id <- 1:15
  out <- women %L>%
    start_log(logger = cellwise$new(key="id")) %L>%
    transform(height = height*0.0254  ) %L>%
    transform(weight = weight*0.453592) %L>%
    transform(bmi    = weight/height^2) %L>%
    dump_log()
  head( read.csv("cellwise.csv"), 3)
```

```
  step                 time srcref                          expression key variable old    new
1    1 2024-09-12 06:06:45 UTC     NA transform(height = height * 0.0254)   1   height  58 1.4732
2    1 2024-09-12 06:06:45 UTC     NA transform(height = height * 0.0254)  10   height  67 1.7018
3    1 2024-09-12 06:06:45 UTC     NA transform(height = height * 0.0254)  11   height  68 1.7272
```

The logging data consists of a step number, a timestamp, the location of the expression in the script (here: `NA`, since there is no script file), the expression that transformed the data, the record key, the variable, the old and the new value.

The variable out contains the output of the calculation.

```
  head(out,3)
```

```
  height   weight id      bmi
1 1.4732 52.16308  1 24.03476
2 1.4986 53.07026  2 23.63087
3 1.5240 54.43104  3 23.43563
```

7 Available loggers

lumberjack is extensible and users can provide their own loggers, for example to offload logging results to a data base or to define new ways to measure the difference between two data sets. Below we list loggers that we know of.

7.1 In the lumberjack package

- `simple` Just check whether data has changed.
- `cellwise` Track changes per cell (incl. old value, new value)
- `filedump` Dump a file after each step (including the zeroth step.)
- `expression_logger` Track the result of any expression.

Both `cellwise` and `simple` have been discussed before. The expression logger tracks the result of one or more expressions that will be evaluated after each data processing step. For example, suppose we want to follow the mean and variance of variables in the 'women' dataset as it gets processed.

```
  logger <- expression_logger$new(mean_h = mean(height), sd_h = sd(height))
  out <- women %L>%
    start_log(logger) %L>%
    transform(height = height*2.54) %L>%
    transform(weight = weight*0.453592) %L>%
    dump_log()
  read.csv("expression.csv",stringsAsFactors = FALSE)
```

```
  step srcref                            expression mean_h     sd_h
1    1     NA     transform(height = height * 2.54)  165.1 11.35923
2    2     NA transform(weight = weight * 0.453592)  165.1 11.35923
```

7.2 In other packages

- `validate::lbj_rules` Track changes in data quality measured by validation rules.
- `validate::lbj_cells` Track changes in cell filling and cell counts.
- `daff::lbj_daff` Use data-diff to track changes in data frame-like objects.

8 Properties of the lumberjack pipe operator

There are several 'forward pipe' operators in the R community, including magrittr, pipeR and yapo. All have different behavior. The lumberjack operator behaves as a simplified version of the 'magrittr' pipe operator. Here are some examples.

```
  # pass the first argument to a function
  1:3 %L>% mean()
```

```
[1] 2
```

```
  # pass arguments using "."
  TRUE %L>% mean(c(1,NA,3), na.rm = .)
```

```
[1] 2
```

```
  # pass arguments to an expression, using "."
  1:3 %L>% { 3 * .}
```

```
[1] 3 6 9
```

```
  # in a more complicated expression, return "." explicitly
  women %L>% { .$height <- 2*.$height; . } %L>% head(3)
```

```
  height weight id
1    116    115  1
2    118    117  2
3    120    120  3
```

The main differences with 'magrittr' are that

- there is no assignment-pipe like %<>%.
- it does not allow you to define functions in the magrittr style: `a <- . %>% sin(.)`
- you cannot do things like `pi %>% sin` and expect an answer.

9 Extending lumberjack

There are many ways to register changes in data. That is why **lumberjack** is extensible with new loggers.

9.1 The lumberjack logging API

In short, a logger is a *reference object* with the following *mandatory* elements:

1. A method `$add(meta, input, output)`. This is a function that computes the difference between `input` and `output` and adds it to a log. The `meta` argument is a `list` with the following elements:
   - `expr` The expression used to turn `input` into `output`.
   - `src` The same expression, but turned into a string.
   - `file` The name of the file that was run. This element is only available when code is run from a script.
   - `lines` A named `integer` vector containing the first and last line of the expression in the source file. This element is only available when code is run from a script.

2. A method `$dump(...)` this function writes the current logging info somewhere. Often this will be a file, but it really can be any place where R can send data. It is *recommended* that `dump` has the argument `file` if it writes anything to file. `$dump` *must* have the `...` argument because when a user calls `dump_log(...)` the extra arguments are passed to `$dump`.
3. a slot called `$label`. The label is set by `start_log` and is used to keep track of logging streams when multiple datasets are tracked with instances of the same logger type. `start_log` will try to create a label if none is provided. If it fails to create a label, it will be set to the empty string `""`.

The following element is *optional*

1. A method `$stop()` called by `stop_log()` before removing the logger from the data.

There are several systems in R to build such a reference object. We recommend using R6 classes or reference classes. Below an example for each system is given. The example loggers only register whether something has ever changed. A `dump` results in a simple message on screen.

9.2 R6 classes

An introduction to R6 classes can be found here.

Let us define the 'trivial' logger.

```
library(R6)
trivial <- R6Class("trivial",
  public = list(
    changed = NULL
  , label=NULL
  , initialize = function(){
      self$changed <- FALSE
  }
  , add = function(meta, input, output){
    self$changed <- self$changed | !identical(input, output)
  }
  , dump = function(){
    msg <- if(self$changed) "" else "not "
    cat(sprintf("The data has %schanged\n",msg))
  }
  )
)
```

Here is how to use it.

```
library(lumberjack)
out <- women %L>%
  start_log(trivial$new()) %L>%
  identity() %L>%
  dump_log(stop=TRUE)
```

```
The data has not changed
```

```
out <- women %L>%
  start_log(trivial$new()) %L>%
  head() %L>%
  dump_log(stop=TRUE)
```

```
The data has changed
```

9.3 Reference classes

Reference classes (RC) come with the R recommended 'methods' package. An introduction can be found here. Here is how to define the trivial logger as a reference class.

```
library(methods)
trivial <- setRefClass("trivial",
  fields = list(
    changed = "logical", label="character"
  ),
  methods = list(
    initialize = function(){
      .self$changed = FALSE
      .self$label = ""
    }
    , add = function(meta, input, output){
      .self$changed <- .self$changed | !identical(input,output)
    }
    , dump = function(){
      msg <- if( .self$changed ) "" else "not "
      cat(sprintf("The data has %schanged\n",msg))
    }
  )
)
```

And here is how to use it.

```
library(lumberjack)
out <- women %L>%
  start_log(trivial()) %L>%
  identity() %L>%
  dump_log(stop=TRUE)
```

```
The data has not changed
```

```
out <- women %L>%
  start_log(trivial()) %L>%
  head() %L>%
  dump_log(stop=TRUE)
```

```
The data has changed
```

Observe that there are subtle differences between R6 and Reference classes (RC).

- In R6 the object is referred to with 'self', in RC this is done with '.self'.
- An R6 object is initialized with `classname$new()`, an RC object is initialized with `classname()`.

9.4 Advice for package authors

If you have a package that has interesting functionality that can be offered also inside a logger, you might consider exporting a logger object that works with **lumberjack**. To keep things uniform, we give the following advice.

Documenting logging objects. Most package authors use roxygen2 to generate documentation. Below is an example of how to document the class and its methods. To show how to document arguments, a new `allcaps`

argument is added to the dump function.

```
#' The trivial logger.
#'
#' The trivial logger only registers whether something has changed at all.
#' A `dump` leads to an informative message on the console.
#'
#' @section Creating a logger:
#' \code{trivial$new()}
#'
#' @section Dump options:
#' \code{$dump(allcaps)}
#' \tabular{ll}{
#'   \code{allcaps}\tab \code{[logical]} print message in capitals?
#' }
#'
#'
#' @docType class
#' @format An \code{R6} class object.
#'
#' @examples
#' out <- women %L>%
#'  start_log(trivial$new()) %L>%
#'  head() %L>%
#'  dump_log(stop=TRUE)
#'
#'
#' @export
trivial <- R6Class("trivial",
  public = list(
    changed = NULL
  , initialize = function(){
      self$changed <- FALSE
  }
  , add = function(meta, input, output){
    self$changed <- self$changed | !identical(input, output)
  }
  , dump = function(allcaps=FALSE){
    msg <- if(self$changed) "" else "not "
    msg <- sprintf("The data has %schanged\n",msg)
    if (allcaps) msg <- toupper(msg)
    cat(msg)
  )
)
```

Adding lumberjack to the DESCRIPTION of your package Once you have exported a logger, it is a good idea to add the line

```
Enhances: lumberjack
```

To the `DESCRIPTION` file. It can then be found by other users via lumberjack's CRAN webpage.