Be wary of silent file replacements!

In base R, you can easily export any object from the environment to an RDS file with:

```
saveRDS(object = my_object, file = "path/to/dir/my_object.RDS")
```

However, including such a line somewhere in your script can carry unintended consequences: When calling `saveRDS` multiple times with identical file names, R *silently overwrites existing, identically named .RDS files* in the specified directory. If the object you are exporting is not what you expect it to be — for example due to some bug in newly edited code — your working copy of the RDS file is simply overwritten in-place. Needless to say, this can prove undesirable.

If you are familiar with this pitfall, you probably used to forestall such potentially troublesome side effects by commenting out the respective lines, then carefully checking each time whether the R object looked fine, then executing the line manually. But even when there is nothing wrong with the R object you seek to export, it can make sense to retain an archived copy of previous RDS files: Think of a dataset you run through a data prep script, and then you get an update of the raw data, or you decide to change something in the data prep (like removing a variable). You may wish to archive an existing copy in such cases, especially with complex data prep pipelines with long execution time.

Don’t get tangled up in manual renaming

You could manually move or rename the existing file each time you plan to create a new one, but that’s tedious, error-prone, and does not allow for unattended execution and scalability. For this reason, I set out to write a carefully designed wrapper function around the existing `saveRDS` call, which is pretty straightforward: As a first step, it checks if the file you attempt to save already exists in the specified location. If it does, the existing file is renamed/archived (with customizable options), and the „updated“ file will be saved under the originally specified name.

This approach has the crucial advantage that the existing code that depends on the file name remaining identical (such as `readRDS` calls in other scripts) will continue to work with the latest version without any needs for adjustment! No more saving your objects as „models_2020-07-12.RDS“, then combing through the other scripts to replace the file name, only to repeat this process the next day. At the same time, an archived copy of the — otherwise overwritten — file will be kept.

What are RDS files anyways?

Before I walk you through my proposed solution, let’s first examine the basics of *serialization*, the underlying process behind high-level functions like `saveRDS`.

> Simply speaking, serialization is the „process of converting an object into a stream of bytes so that it can be transferred over a network or stored in a persistent storage.“
>
> *Stack Overflow: What is serialization?*

There is also a low-level R interface, `serialize`, which you can use to explore (un-)serialization first-hand: Simply fire up R and run something like `serialize(object = c(1, 2, 3), connection = NULL)`. This call serializes the specified vector and prints the output right to the console. The result is an odd-looking *raw vector*, with each byte separately represented as a pair of hex digits. Now let’s see what happens if we revert this process:

```
s <- serialize(object = c(1, 2, 3), connection = NULL)
print(s)
# >  [1] 58 0a 00 00 00 03 00 03 06 00 00 03 05 00 00 00 00 05 55 54 46 2d 38 00
00 00 0e 00
# > [29] 00 00 03 3f f0 00 00 00 00 00 00 40 00 00 00 00 00 00 00 40 08 00 00 00
00 00 00
```

```
unserialize(s)
# > 1 2 3
```

The length of this raw vector increases rapidly with the complexity of the stored information: For instance, serializing the famous, although not too large, `iris` dataset results in a raw vector consisting of 5959 pairs of hex digits!

Besides the already mentioned `saveRDS` function, there is also the more generic `save` function. The former saves a *single* R object to a file. It allows us to restore the object from that file (with the counterpart `readRDS`), possibly under a different variable name: That is, you can *assign* the contents of a call to `readRDS` to another variable. By contrast, `save` allows for saving *multiple* R objects, but when reading back in (with `load`), they are simply restored in the environment under the object names they were saved with. (That's also what happens automatically when you answer „Yes" to the notorious question of whether to „save the workspace image to ~/.RData" when quitting RStudio.)

Creating the archives

Obviously, it's great to have the possibility to save internal R objects to a file and then be able to re-import them in a clean session or on a different machine. This is especially true for the results of long and computationally heavy operations such as fitting machine learning models. But as we learned earlier, one wrong keystroke can potentially erase that one precious 3-hour-fit fine-tuned XGBoost model you ran and carefully saved to an RDS file yesterday.

Digging into the wrapper

So, how did I go about fixing this? Let's take a look at the code. First, I define the arguments and their defaults: The `object` and `file` arguments are taken directly from the wrapped function, the remaining arguments allow the user to customize the archiving process: Append the archive file name with either the date the original file was archived or last modified, add an additional timestamp (not just the calendar date), or save the file to a dedicated archive directory. For more details, please check the documentation here. I also include the ellipsis `...` for additional arguments to be passed down to `saveRDS`. Additionally, I do some basic input handling (not included here).

```
save_rds_archive <- function(object,
                             file = "",
                             archive = TRUE,
                             last_modified = FALSE,
                             with_time = FALSE,
                             archive_dir_path = NULL,
                             ...) {
```

The main body of the function is basically a series of if/else statements. I first check if the `archive` argument (which controls whether the file should be archived in the first place) is set to `TRUE`, and then if the file we are trying to save already exists (note that „file“ here actually refers to the whole file *path*). If it does, I call the internal helper function `create_archived_file`, which eliminates redundancy and allows for concise code.

```
if (archive) {

    # check if file exists
    if (file.exists(file)) {

      archived_file <- create_archived_file(file = file,
                                            last_modified = last_modified,
                                            with_time = with_time)
```

Composing the new file name

In this function, I create the new name for the file which is to be archived, depending on user input: If `last_modified` is set, then the `mtime` of the file is accessed. Otherwise, the current system date/time (= the date of archiving) is taken instead. Then the spaces and special characters are replaced with underscores, and, depending on the value of the `with_time` argument, the actual *time* information (not just the calendar date) is kept or not.

To make it easier to identify directly from the file name what exactly (date of archiving vs. date of modification) the indicated date/time refers to, I also add appropriate information to the file name. Then I save the file extension for easier replacement (note that „.RDS“, „.Rds“, and „.rds“ are all valid file extensions for RDS files). Lastly, I replace the current file extension with a concatenated string containing the type info, the new date/time suffix, and the original file extension. Note here that I add a „$“ sign to the regex which is to be matched by `gsub` to only match the *end* of the string: If I did not do that and the file name would be something like „my_RDS.RDS“, then both matches would be replaced.

```
# create_archived_file.R

create_archived_file <- function(file, last_modified, with_time) {

  # create main suffix depending on type
  suffix_main <- ifelse(last_modified,
                        as.character(file.info(file)$mtime),
                        as.character(Sys.time()))

  if (with_time) {

    # create clean date-time suffix
    suffix <- gsub(pattern = " ", replacement = "_", x = suffix_main)
    suffix <- gsub(pattern = ":", replacement = "-", x = suffix)

    # add "at" between date and time
    suffix <- paste0(substr(suffix, 1, 10), "_at_", substr(suffix, 12, 19))

  } else {

    # create date suffix
    suffix <- substr(suffix_main, 1, 10)

  }

  # create info to paste depending on type
  type_info <- ifelse(last_modified,
```

```
                        "_MODIFIED_on_",
                        "_ARCHIVED_on_")

  # get file extension (could be any of "RDS", "Rds", "rds", etc.)
  ext <- paste0(".", tools::file_ext(file))

  # replace extension with suffix
  archived_file <- gsub(pattern = paste0(ext, "$"),
                        replacement = paste0(type_info,
                                             suffix,
                                             ext),
                        x = file)

  return(archived_file)

}
```

Archiving the archives?

By way of example, with `last_modified = FALSE` and `with_time = TRUE`, this function would turn the character file name „models.RDS“ into „models_ARCHIVED_on_2020-07-12_at_11-31-43.RDS“. However, this is just a character vector for now — the file itself is not renamed yet. For this, we need to call the base R `file.rename` function, which provides a direct interface to your machine's file system. I first check, however, whether a file with the same name as the newly created archived file string already exists: This could well be the case if one appends only the date (`with_time = FALSE`) and calls this function several times per day (or potentially on the same file if `last_modified = TRUE`).

Somehow, we are back to the old problem in this case. However, I decided that it was not a good idea to archive files that are themselves archived versions of another file since this would lead to too much confusion (and potentially too much disk space being occupied). Therefore, only the most recent archived version will be kept. (Note that if you still want to keep multiple archived versions of a single file, you can set `with_time = TRUE`. This will append a timestamp to the archived file name up to the second, virtually eliminating the possibility of duplicated file names.) A warning is issued, and then the already existing archived file will be overwritten with the current archived version.

The last puzzle piece: Renaming the original file

To do this, I call the `file.rename` function, renaming the „file“ originally passed by the user call to the string returned by the helper function. The `file.rename` function always returns a boolean indicating if the operation succeeded, which I save to a variable `temp` to inspect later. Under some circumstances, the renaming process may fail, for instance due to missing permissions or OS-specific restrictions. We did set up a CI pipeline with *GitHub Actions* and continuously test our code on Windows, Linux, and MacOS machines with different versions of R. So far, we didn't run into any problems. Still, it's better to provide in-built checks.

It's an error! Or is it?

The problem here is that, when renaming the file on disk failed, `file.rename` raises merely a *warning*, not an *error*. Since any causes of these warnings most likely originate from the local file system, there is no sense in continuing the function if the renaming failed. That's why I wrapped it into a `tryCatch` call that captures the warning message and passes it to the `stop` call, which then terminates the function with the appropriate message.

Just to be on the safe side, I check the value of the `temp` variable, which should be `TRUE` if the renaming succeeded, and also check if the archived version of the file (that is, the result of our renaming operation) exists. If both of these conditions hold, I simply call `saveRDS` with the original specifications (now that our existing copy has been renamed, nothing will be overwritten if we save the new file with the original name), passing along further arguments with `...`.

```
        if (file.exists(archived_file)) {
          warning("Archived copy already exists - will overwrite!")
        }

        # rename existing file with the new name
        # save return value of the file.rename function
        # (returns TRUE if successful) and wrap in tryCatch
        temp <- tryCatch({file.rename(from = file,
                                      to = archived_file)
        },
        warning = function(e) {
          stop(e)
        })

      }

      # check return value and if archived file exists
      if (temp & file.exists(archived_file)) {
        # then save new file under specified name
        saveRDS(object = object, file = file, ...)
      }

    }
```

These code snippets represent the cornerstones of my function. I also skipped some portions of the source code for reasons of brevity, chiefly the creation of the „archive directory“ (if one is specified) and the process of copying the archived file into it. Please refer to our GitHub for the complete source code of the main and the helper function.

Finally, to illustrate, let’s see what this looks like in action:

```
x <- 5
y <- 10
z <- 20

## save to RDS
saveRDS(x, "temp.RDS")
saveRDS(y, "temp.RDS")

## "temp.RDS" is silently overwritten with y
## previous version is lost
readRDS("temp.RDS")
#> [1] 10

save_rds_archive(z, "temp.RDS")
## current version is updated
readRDS("temp.RDS")
#> [1] 20

## previous version is archived
readRDS("temp_ARCHIVED_on_2020-07-12.RDS")
#> [1] 10
```