**Introduction**

Caching intermediate objects in R can be an efficient way to avoid re-evaluating long-running computations. The general process is always the same: run the chunk of code once, store the output to disk, and load it up the next time the same chunk is run. There are, of course, multiple packages in R to help with this process, so I’ve decided to outline some of the more popular options below.

One of the most important features of any caching system is its ability to detect if the cache has become “stale,” that is, when the object on disk is no longer valid because the dependencies of the cached object have changed. This feature is specifically discussed in the sections for each caching method, but, briefly, **there are systems for *cache invalidation* in R Markdown, ‘R.cache’, ‘mustashe,’ and ‘ProjectTemplate.’**

**Options**

Here are the options for caching in R that I will discuss below, and each has a link to more information on that specific option:

* [in R Markdown](https://bookdown.org/yihui/rmarkdown-cookbook/cache.html)
* [‘memoise’](https://github.com/r-lib/memoise)
* [‘R.cache’](https://cran.r-project.org/web/packages/R.cache/R.cache.pdf)
* [‘mustashe’](https://jhrcook.github.io/mustashe/)
* [‘DataCache’](https://www.r-bloggers.com/data-caching/)
* [‘ProjectTemplate’](http://projecttemplate.net/caching.html)

**TL;DR**

**Caching a code chunk in R Markdown**

R Markdown has a built-in caching feature that can be enabled by setting cache=TRUE in the chunk’s header.

```{r import-df, cache=TRUE}

df <- read\_tsv("data-file.tsv")

```

The second time the chunk is run, both the visual output and any objects created are loaded from disk. If you are already using R Markdown for your project or work, this is probably the only caching mechanism you will need.

R Markdown does have a method for detecting cache invalidation, though it is not *explicitly* supported by ‘knitr.’ The basic idea is to set another chunk option that computes some value that, if it changes, should trigger cache invalidation. For instance, say we are reading in a file from disk and want the chunk to re-run if it changes. We can create a new chunk option called cache.extra and assign it some value to indicate if the file has changed, such as the modification date.

```{r import-df, cache=TRUE, cache.extra=file.mtime("data-file.tsv")}

df <- read\_tsv("data-file.tsv")

```

Now if the file is modified, the cache for the code chunk will be invalidated and the code will be re-run.

**‘memoise’**

The ‘memoise’ package brings in the function memoise(). When a function is “memoised,” the inputs and outputs are remembered so that if a function is passed the same inputs multiple times, the previously computed output can be returned immediately, without re-evaluating the function call. This is an optimization technique from [*dynamic programming*](https://en.wikipedia.org/wiki/Dynamic_programming).

The memoise() function is passed a function and returns a new function with the same properties as the original, except it is now memoised (it returns TRUE when passed to is.memoised()). Below is an example where sq(), a simple function that squares its input, is memoised as memo\_sq(). A print statement is included in the sq() function to indicate when it has actually been run.

library(memoise)

sq <- function(x) {

print("Computing square of 'x'")

x\*\*2

}

memo\_sq <- memoise(sq)

The first time memo\_sq(2) is run, the function is evaluated and we see the print statement’s message.

memo\_sq(2)

#> [1] "Computing square of 'x'"

#> [1] 4

However, the second time, the result is loaded from disk and we see no message.

memo\_sq(2)

#> [1] 4

Optionally, a local directory, AWS S3 bucket, or Google Cloud Storage location can be passed as the location to save the cached data (i.e. paired inputs and outputs). This can be useful for storing the memoised values across multiple R sessions.

As far as I am aware, there is no cache invalidation feature in the ‘memoise’ package. In other words, if I were to change sq() to return the cube of the input, memo\_sq() would *not* be automatically updated or alerted in any way.

sq <- function(x) {

x\*\*3

}

sq(2)

#> [1] 8

memo\_sq(2)

#> [1] 4

In fairness, caching is not the intended purpose of memoisation, but it is a practical use case, so I think it is still worth including in this article.

**‘R.cache’**

The documentation for ‘R.cache’ is limited, but from what I can figure out, it implements memoisation while also linking to dependencies for cache invalidation. Further, and the main distinguishing feature between this package and ‘memoise’, ‘R.cache’ memoises an *expression*, not just a function.

The primary function of ‘R.cache’ is evalWithMemoization(). It takes an expression to be evaluated, evaluates the expression, and stores both the created object, a in this case, and the expression itself.

suppressPackageStartupMessages(library(R.cache))

evalWithMemoization({

print("Evaluating expression.")

a <- 1

})

#> [1] "Evaluating expression."

#> [1] 1

a

#> [1] 1

Now the second time the expression is evaluated, there is no print message because the result is loaded from disk.

library(R.cache)

evalWithMemoization({

print("Evaluating expression.")

a <- 1

})

#> [1] 1

Dependencies can be declared for the memoised expression by passing one or more objects to the key parameter. For example, the object b is listed as a key for the following expression.

b <- 1

evalWithMemoization(

{

print("Evaluating expression.")

a <- 100 + b

},

key = b

)

#> [1] "Evaluating expression."

#> [1] 101

If b doesn’t change, then the expression is not re-evaluated.

evalWithMemoization(

{

print("Evaluating expression.")

a <- 100 + b

},

key = b

)

#> [1] 101

However, if b changes, then the expression is evaluated again.

b <- 2

evalWithMemoization(

{

print("Evaluating expression.")

a <- 100 + b

},

key = b

)

#> [1] "Evaluating expression."

#> [1] 102

While this package has many desirable features for caching, there are some design choices that I do not like. To begin, I am not a huge fan of this package’s API including the function naming scheme and how the keys are passed after the expression. Further, I do not like how the final result of the expression is automatically returned, I would prefer this be returned invisibly if anything. Also, I don’t like that the default location for the caching directory is /Users/admin/Library/Caches/R/R.cache, I would prefer it be a hidden directory in the project’s root directory. Finally, the evaluated expression is not invariant to stylistic changes to the expression. For instance, if the assignment arrow <- is changed to an =, the expression is re-evaluated.

evalWithMemoization({

print("Evaluating expression.")

a = 1

})

#> [1] "Evaluating expression."

#> [1] 1

For these reasons, I created the ‘mustashe’ package, demonstrated next.

**‘mustashe’**

I have recently described ‘mustashe’ in two previous posts, so I will keep the description here brief.

The stash() function takes a name of the stashed value, an expression to evaluate, and any dependencies.

library(mustashe)

x <- 1

stash("y", depends\_on = "x", {

print("Calculating 'y'")

y <- x + 1

})

#> Updating stash.

#> [1] "Calculating 'y'"

# Value of `y`

y

#> [1] 2

Just like ‘R.cache,’ if the value of the dependency x changes, then the code is re-evaluated.

# Change the value of a dependency of `y`.

x <- 2

stash("y", depends\_on = "x", {

print("Calculating 'y'")

y <- x + 1

})

#> Updating stash.

#> [1] "Calculating 'y'"

However, ‘mustashe’ handles stylistic changes to the expression better than ‘R.cache’. For instance, if the same code was instead typed by a madman, ‘mustashe’ would still not re-run the code chunk.

stash("y", depends\_on = "x", {

print( "Calculating 'y'" )

y = x + 1

# Add a new comment!

})

#> Loading stashed object.

Overall, ‘mustashe’ and ‘R.cache’ are very similar, and the main differences are stylistic.

**‘DataCache’**

I won’t discuss the ‘DataCache’ package extensively because I personally have little use for it. It has already been explained by the author on a previous R-Blogger’s post, [‘Data Caching’](https://www.r-bloggers.com/data-caching/), so if you are interested, I recommend reading that article. Also, it is not on CRAN nor actively maintained on GitHub. In general it is intended to periodically load data from an external source. The idea is the the data is dynamic and frequently updated. The ‘DataCache’ package sets a timer for the data and reads in the most recent version at set periods.

**‘ProjectTemplate’**

The ‘ProjectTemplate’ package is *far* more than a caching system, rather, it is a data analysis project framework. The caching system is merely a part of it. However, the entire framework must be adopted in order to use its caching system. For this reason, I will not provide an in depth preview of their system, but just provide the following example. (Note, the API is very similar to that used by ‘mustashe’ because it was the inspiration for that package.)

cache("foo", depends = c("a", "b"), {

x <- loaded\_data$name

x <- as.character(x)

c(x[[1]], a, b)

})

**Conclusion**

Here are my recommendations for what caching system to use, in order of precedence:

1. If you just want memoisation for its intended purpose (i.e. avoid repetitive calculations), use the ‘memosie’ package.
2. If using the ‘ProjectTemplate’ framework, then use its built in caching system.
3. If you are using an R Markdown file, then use the chunk caching feature.
4. For all other caching needs, choose between ‘mustashe’ and ‘R.cache’ (I prefer using ‘mustashe’, but I am biased).