**Named Map Builder**

First I will demonstrate wrapr‘s "named map builder": :=. The named map builder adds names to vectors and lists by nice "names on the left and values on the right" notation.

Set names of right-argument to be left-argument, and return right argument. Called from := operator.

named\_map\_builder(targets, values)

`:=`(targets, values)

targets %:=% values

For example to build a named vector mapping names c("a", "b") to values c(1, 2) we could write the following R code.

c(a = 1, b = 2)

## a b

## 1 2

Or we can write:

c("a" = 1, "b" = 2)

## a b

## 1 2

Using wrapr we can write the same thing for quoted names using :=.

library("data.table") # data.table before wrapr to avoid := contention

suppressPackageStartupMessages(library("dplyr"))

library("wrapr")

##

## Attaching package: 'wrapr'

## The following object is masked from 'package:data.table':

##

## :=

c("a" := 1, "b" := 2)

## a b

## 1 2

This is read as "a is 1 and b is 2".

So far, no gain (in fact it has forced some quotes on us). However (unlike = and <-), := also works vectorized (as shown below).

c("a", "b") := c(1, 2)

## a b

## 1 2

:= works the same with variable as it does with values:

names <- c("a", "b")

values <- c(1, 2)

names := values

## a b

## 1 2

This above notation is the usual use of :=.

One can think of := as an operator version of setNames(nm = c("a", "b"), c(1,2)) (from stats). This notation is very handy once you look for places to use it and for tools to further neaten it up. I recommend using binding the := glyph to the key chord "Alt-=" in RStudio using the addinexamplesWV package.

**Quoting Combine**

qc() (quoting combine/concatenate) is another wrapr notation improving function. qc() work by analogy to R‘s c() function, except it quotes its arguments. This lets us write the previous vector naming as:

qc(a, b) := c(1, 2)

## a b

## 1 2

**let()**

The above notations work particularly well with wrapr::let().

wrapr::let() evaluates an expression (or block of expressions) with a number of symbolic substitutions. The named map builder is a great way to specify such substitutions. A quick examples is computing "1 plus variable" where the actual name of the variable is specified in a named vector called "mapping".

mapping <- "VARNAME" := "x"

x <- 1 # the actual varaible

# Evaluate "VARNAME + 1" with mapping substitution

let(alias = mapping,

VARNAME + 1

)

## [1] 2

The mapping from abstract variables (variable names used in the code) to concrete variables (variable names with desired values in the execution environment) can also be performed inline:

let("VARNAME" := "x",

VARNAME + 1

)

## [1] 2

let() itself is useful in writing re-usable (or parametric) functions (often a challenge in R).

d <- data.frame(x = c(1, 2))

incrementColumn <- function(data, COLUMNNAME) {

let(c("COLUMNNAME" = COLUMNNAME),

mutate(data, COLUMNNAME = COLUMNNAME + 1)

)

}

incrementColumn(d, "x")

## x

## 1 2

## 2 3

The idea is we use the stand-in symbol COLUMNNAME in our code (no matter how complicated) and let() substitutes name (represented as a string) stored into the expression before execution. What we just executed was equivalent to the following:

COLUMNNAME = "x"

let(c("COLUMNNAME" = COLUMNNAME), eval = FALSE,

mutate(d, COLUMNNAME = COLUMNNAME + 1)

)

## mutate(d, x = x + 1)

The commonly suggested method of performing these substitutions in dplyr without wrapr (I strongly prefer using wrapr) is:

COLUMNSYM = rlang::sym("x")

mutate(d, !!COLUMNSYM := (!!COLUMNSYM) + 1)

## x

## 1 2

## 2 3

**mapsyms() (the let(X=X), replace with values convention)**

wrapr::mapsyms() is a helper function makes function creation even more convenient. A mapsyms expression of the form mapsyms(COLUMNNAME) is equivalent to the code c("COLUMNNAME" = COLUMNNAME). In our example that means it builds the name to name mapping: c(‘COLUMNNAME’ = ‘x’) (here we used wrapr::map\_to\_char() to present the result). With mapsyms() we can write the earlier function as:

Library(wrapr)

incrementColumn <- function(data, COLUMNNAME) {

let(mapsyms(COLUMNNAME),

mutate(data, COLUMNNAME = COLUMNNAME + 1)

)

}

incrementColumn(d, "x")

## x

## 1 2

## 2 3

I have more mapsyms() examples in our article "Let X=X in R":

The idea was that perhaps one had worked out a complicated (but useful and important) by-hand survey scoring method:

suppressPackageStartupMessages(library("dplyr"))

library("wrapr")

d <- data.frame(

subjectID = c(1,

1,

2,

2),

surveyCategory = c(

'withdrawal behavior',

'positive re-framing',

'withdrawal behavior',

'positive re-framing'

),

assessmentTotal = c(5,

2,

3,

4),

stringsAsFactors = FALSE

)

scale <- 0.237

d %>%

group\_by(subjectID) %>%

mutate(probability =

exp(assessmentTotal \* scale)/

sum(exp(assessmentTotal \* scale))) %>%

arrange(probability, surveyCategory) %>%

mutate(isDiagnosis = row\_number() == n()) %>%

filter(isDiagnosis) %>%

ungroup() %>%

select(subjectID, surveyCategory, probability) %>%

rename(diagnosis = surveyCategory) %>%

arrange(subjectID)

## # A tibble: 2 x 3

## subjectID diagnosis probability

## <dbl> <chr> <dbl>

## 1 1 withdrawal behavior 0.6706221

## 2 2 positive re-framing 0.5589742

The presumption is that the above pipeline is considered reasonable (but long, complicated, and valuable) dplyr, and our goal is to re-use it on new data that may not have the same column names as our original data.

We are making the huge simplifying assumption that you have studied the article and the above example is now familiar.

The question is: what to do when one wants to process the same type of data with different column names? For example:

d <- data.frame(

PID = c(1,

1,

2,

2),

DIAG = c(

'withdrawal behavior',

'positive re-framing',

'withdrawal behavior',

'positive re-framing'

),

AT = c(5,

2,

3,

4),

stringsAsFactors = FALSE

)

print(d)

## PID DIAG AT

## 1 1 withdrawal behavior 5

## 2 1 positive re-framing 2

## 3 2 withdrawal behavior 3

## 4 2 positive re-framing 4

The new table has the following new column definitions:

subjectID <- "PID"

surveyCategory <- "DIAG"

assessmentTotal <- "AT"

isDiagnosis <- "isD"

probability <- "prob"

diagnosis <- "label"

We could "reduce to a previously solved problem" by renaming the columns to names we know, doing the work, and then renaming back (which is actually a service that replyr::replyr\_apply\_f\_mapped() supplies).

I advised editing the pipeline to have obvious stand-in names (perhaps in all-capitals) and then using wrapr::let() to perform symbol substitution on the pipeline.

Let’s try that:

let(

c(subjectID = subjectID,

surveyCategory = surveyCategory,

assessmentTotal = assessmentTotal,

isDiagnosis = isDiagnosis,

probability = probability,

diagnosis = diagnosis),

d %>%

group\_by(subjectID) %>%

mutate(probability =

exp(assessmentTotal \* scale)/

sum(exp(assessmentTotal \* scale))) %>%

arrange(probability, surveyCategory) %>%

mutate(isDiagnosis = row\_number() == n()) %>%

filter(isDiagnosis) %>%

ungroup() %>%

select(subjectID, surveyCategory, probability) %>%

rename(diagnosis = surveyCategory) %>%

arrange(subjectID))

## # A tibble: 2 x 3

## PID label prob

## <dbl> <chr> <dbl>

## 1 1 withdrawal behavior 0.6706221

## 2 2 positive re-framing 0.5589742

That works! All we did was: paste the original code into the block and the adapter did all of the work, with no user edits of the code.

It is a bit harder for the user to find which symbols are being replaced, but in some sense they don’t really need to know (it is R‘s job to perform the replacements).

wrapr has a new helper function mapsyms() that automates all of the "let x = x" steps from the above example.

mapsyms() is a simple function that captures variable names and builds a mapping from them to the names they refer to in the current environment. For example we can use it to quickly build the assignment map for the let block, because the earlier assignments such as "subjectID <- "PID"" allow mapsyms() to find the intended re-mappings. This would also be true for other cases, such as re-mapping function arguments to values. Our example becomes:

print(mapsyms(subjectID,

surveyCategory,

assessmentTotal,

isDiagnosis,

probability,

diagnosis))

## $subjectID

## [1] "PID"

##

## $surveyCategory

## [1] "DIAG"

##

## $assessmentTotal

## [1] "AT"

##

## $isDiagnosis

## [1] "isD"

##

## $probability

## [1] "prob"

##

## $diagnosis

## [1] "label"

This allows the solution to be re-written and even wrapped into a function in a very legible form with very little effort:

computeRes <- function(d,

subjectID,

surveyCategory,

assessmentTotal,

isDiagnosis,

probability,

diagnosis) {

let(

mapsyms(subjectID,

surveyCategory,

assessmentTotal,

isDiagnosis,

probability,

diagnosis),

d %>%

group\_by(subjectID) %>%

mutate(probability =

exp(assessmentTotal \* scale)/

sum(exp(assessmentTotal \* scale))) %>%

arrange(probability, surveyCategory) %>%

mutate(isDiagnosis = row\_number() == n()) %>%

filter(isDiagnosis) %>%

ungroup() %>%

select(subjectID, surveyCategory, probability) %>%

rename(diagnosis = surveyCategory) %>%

arrange(subjectID)

)

}

computeRes(d,

subjectID = "PID",

surveyCategory = "DIAG",

assessmentTotal = "AT",

isDiagnosis = "isD",

probability = "prob",

diagnosis = "label")

## # A tibble: 2 x 3

## PID label prob

## <dbl> <chr> <dbl>

## 1 1 withdrawal behavior 0.6706221

## 2 2 positive re-framing 0.5589742

The idea is: instead of having to mark what instances of symbols are to be replaced (by quoting or de-quoting indicators), we instead declare what symbols are to be replaced using the mapsyms() helper.

The let() method of building functions works well with dplyr and data.table. For each of these let’s show code for the "by hand logistic scoring" example :

For example, our scientist might record the results of two subjects taking a test as follows:

d <- data.frame(

subjectID = c(1,

1,

2,

2),

surveyCategory = c(

'withdrawal behavior',

'positive re-framing',

'withdrawal behavior',

'positive re-framing'

),

assessmentTotal = c(5,

2,

3,

4),

stringsAsFactors = FALSE

)

print(d)

## subjectID surveyCategory assessmentTotal

## 1 1 withdrawal behavior 5

## 2 1 positive re-framing 2

## 3 2 withdrawal behavior 3

## 4 2 positive re-framing 4

# or in "wide form":

library("cdata")

moveValuesToColumns(d,

columnToTakeKeysFrom = 'surveyCategory',

columnToTakeValuesFrom = 'assessmentTotal',

rowKeyColumns = 'subjectID')

## subjectID positive re-framing withdrawal behavior

## 1 1 2 5

## 2 2 4 3

A natural question is: how does one assign weights to each answer? One way would be to administer the test to a number of people the experimenter has classified as having either of the above mentioned behaviors and then performing a logistic regression to map assessment answers to the probability of a given diagnosis for this population. By re-scaling the weights and rounding them to small integers we could have a test point system that is very close to performing a logistic regression classification. We may be able to use the same assessment questions in a much more decisive manner than assigning all questions the same number of points.

This sort of idea is what one would expect from a mixed and collaborating team that includes medical experts, statistics experts, and programmers. After some work our team might work out that scoring the assessment can be done by the simple R dplyr pipeline:

suppressPackageStartupMessages(library("dplyr"))

scale <- 0.237

d %>%

group\_by(subjectID) %>%

mutate(probability =

exp(assessmentTotal \* scale)/

sum(exp(assessmentTotal \* scale)))

## # A tibble: 4 x 4

## # Groups: subjectID [2]

## subjectID surveyCategory assessmentTotal probability

## <dbl> <chr> <dbl> <dbl>

## 1 1 withdrawal behavior 5 0.6706221

## 2 1 positive re-framing 2 0.3293779

## 3 2 withdrawal behavior 3 0.4410258

## 4 2 positive re-framing 4 0.5589742

For each subject we take the row with maximal probability as the diagnosis. The diagnosis was already obvious from the original scores, the main addition is the diagnosis confidence is now available as a probability estimate.

Each step of the above pipeline is learn-able:

* The group\_by() is arranging all rows associated with the same subject to work together in later calculations.
* the exp(assessmentTotal \* scale)/sum(exp(assessmentTotal \* scale)) is the classic "sigmoid link" from logistic regression. It is the standard way (once you know it) of turning a free-score into a probability estimate.

Suppose this assessment is tested and works well. It is then plausible that the team might ask their R expert to help them construct a much more complicated dplyr pipeline that better formats the results. Under the Harlan Mills’ "Surgical Team" proposal we expect effective data science teams to have a diversity of deep expertise (not everybody know everything, but a lot is known by the total team). We expect a well staffed research team to include the statistician who worked out the sigmoid transform above, and a programmer who works out the pipeline we give below.

d %>%

group\_by(subjectID) %>%

mutate(probability =

exp(assessmentTotal \* scale)/

sum(exp(assessmentTotal \* scale))) %>%

arrange(probability, surveyCategory) %>%

mutate(isDiagnosis = row\_number() == n()) %>%

filter(isDiagnosis) %>%

ungroup() %>%

select(subjectID, surveyCategory, probability) %>%

rename(diagnosis = surveyCategory) %>%

arrange(subjectID)

## # A tibble: 2 x 3

## subjectID diagnosis probability

## <dbl> <chr> <dbl>

## 1 1 withdrawal behavior 0.6706221

## 2 2 positive re-framing 0.5589742

This is indeed a long (and expert-level) pipeline. But the principle is:

* It does useful work (concentrates down to the rows we want and ensures good presentation column names and sorting).
* While a part-time R user would not be expected to come up with it, they could (with cooperation from the pipeline author) understand all the steps and safely use the pipeline in their project.
* The application (which we spent some time describing) is what the team cares about, the pipeline is a ends to a means (so even though it is long, it isn’t often the central subject of interest).
* The longer pipeline is paying the bills, and helping patients. So some pain and cost are to be tolerated.

Let’s take this deliberately long (so as to be a strong test) example and see how hard the pipeline is to re-use under different methodologies.

**Re-use**

An issue that comes up is: can the team re-use the pipeline on another project? Suppose in their next project the ID column isn’t "subjectID" but it is "patientID" (and so on). Obviously they can copy and paste the original pipeline and change the names (which is not a bad practice for the first few re-uses).

But once this procedure is going to be used many times it is a good idea to wrap it up or genericize it so it can be safely re-adapted (so the users can’t accidentally forget to change one name one place).

I will now walk through a number of approaches to this in terms of how hard they are on the researcher. We are assuming their R expert does the wrapping for them, but then must explain the concepts to the part-time R user so they truly understand and can maintain the tools they are using.

For our example we assume all the column names are coming from variables set somewhere else (in another R script, or coming from a spreadsheet that is read into R, or some other source). The nature of the columns is constant from analysis to analysis, but the exact names used may vary. For our example the column names are:

idCol <- "subjectID"

categoryCol <- "surveyCategory"

linkScoreCol <- "assessmentTotal"

indicatorCol <- "isDiagnosis"

probScoreCol <- "probability"

outcomeCol <- "diagnosis"

**wrapr solution**

In my opinion the easiest solution (in terms of cognitive load) is wrapr::let(). The R expert would share the following code:

library("wrapr")

let(

c(

IDCOL = idCol,

CATEGORYCOL = categoryCol,

LINKSCORECOL = linkScoreCol,

INDICATORCOL = indicatorCol,

PROBSCORECOL = probScoreCol,

OUTCOMECOL = outcomeCol

),

d %>%

group\_by(IDCOL) %>%

mutate(PROBSCORECOL =

exp(LINKSCORECOL \* scale)/

sum(exp(LINKSCORECOL \* scale))) %>%

arrange(PROBSCORECOL, CATEGORYCOL) %>%

mutate(INDICATORCOL = row\_number() == n()) %>%

filter(INDICATORCOL) %>%

ungroup() %>%

select(IDCOL, CATEGORYCOL, PROBSCORECOL) %>%

rename(OUTCOMECOL = CATEGORYCOL) %>%

arrange(IDCOL)

)

## # A tibble: 2 x 3

## subjectID diagnosis probability

## <dbl> <chr> <dbl>

## 1 1 withdrawal behavior 0.6706221

## 2 2 positive re-framing 0.5589742

The concept is:

"let() works as if you had written the code with the names substituted as shown in the c() block."

And there is ample documentation showing how this can be used. Notice creating this code is completely mechanical (replace concrete names with the all-caps place holders) and the execution has an easy mental model (the place-holders are replaced with names stored in the variables).

In this solution the adapted code looks like the original code.

**replyr solution**

The next easiest method *in concept* is replyr\_apply\_f\_mapped().

The R expert would write the following, and the part-time R user (with some coaching) could maintain it.

library("replyr")

d %>%

replyr\_apply\_f\_mapped(

nmap = c(

IDCOL = idCol,

CATEGORYCOL = categoryCol,

LINKSCORECOL = linkScoreCol,

INDICATORCOL = indicatorCol,

PROBSCORECOL = probScoreCol,

OUTCOMECOL = outcomeCol

),

f = . %>%

group\_by(IDCOL) %>%

mutate(PROBSCORECOL =

exp(LINKSCORECOL \* scale)/

sum(exp(LINKSCORECOL \* scale))) %>%

arrange(PROBSCORECOL, CATEGORYCOL) %>%

mutate(INDICATORCOL = row\_number() == n()) %>%

filter(INDICATORCOL) %>%

ungroup() %>%

select(IDCOL, CATEGORYCOL, PROBSCORECOL) %>%

rename(OUTCOMECOL = CATEGORYCOL) %>%

arrange(IDCOL)

)

## # A tibble: 2 x 3

## subjectID diagnosis probability

## <dbl> <chr> <dbl>

## 1 1 withdrawal behavior 0.6706221

## 2 2 positive re-framing 0.5589742

What the code does is exactly this:

* It renames all of the columns in the data.frame to have the chosen names (in this case the all-caps names).
* It then applies the user-supplied function f() to this data.frame.
* The reverse of the name-mapping is applied to the result of f(), moving columns back to their original names.

The concept is:

replyr\_apply\_f\_mapped() renames columns and back.

Below is an illustrative example showing the column names seen inside and outside the user supplied function.

print(colnames(d))

## [1] "subjectID" "surveyCategory" "assessmentTotal"

d %>%

replyr\_apply\_f\_mapped(

nmap = c(

IDCOL = idCol,

CATEGORYCOL = categoryCol,

LINKSCORECOL = linkScoreCol,

INDICATORCOL = indicatorCol,

PROBSCORECOL = probScoreCol,

OUTCOMECOL = outcomeCol

),

f = function(df) {

df$PROBSCORECOL <- 1

print(colnames(df))

return(df)

}

) %>%

colnames()

## [1] "IDCOL" "CATEGORYCOL" "LINKSCORECOL" "PROBSCORECOL"

## [1] "subjectID" "surveyCategory" "assessmentTotal" "probability"

This is teachable and something the part-time R user can correctly extend and maintain. Though the user may possibly need to learn about wrapping a pipeline as an anonymous function (the ". %>%" notation).

**rlang/tidyeval solution**

For the rlang/tidyeval solution the expert writes the following code:

IDSYM <- rlang::sym(idCol)

CATEGORYSYM <- rlang::sym(categoryCol)

LINKSCORESYM <- rlang::sym(linkScoreCol)

INDICATORSYM <- rlang::sym(indicatorCol)

PROBSCORESYM <- rlang::sym(probScoreCol)

OUTCOMESYM <- rlang::sym(outcomeCol)

d %>%

group\_by(!!IDSYM) %>%

mutate(!!PROBSCORESYM :=

exp((!!LINKSCORESYM) \* scale)/

sum(exp((!!LINKSCORESYM) \* scale))) %>%

arrange(!!PROBSCORESYM, !!CATEGORYSYM) %>%

mutate(!!INDICATORSYM := row\_number() == n()) %>%

filter(!!INDICATORSYM) %>%

ungroup() %>%

select(!!IDSYM, !!CATEGORYSYM, !!PROBSCORESYM) %>%

rename(!!OUTCOMESYM := !!CATEGORYSYM) %>%

arrange(!!IDSYM)

## # A tibble: 2 x 3

## subjectID diagnosis probability

## <dbl> <chr> <dbl>

## 1 1 withdrawal behavior 0.6706221

## 2 2 positive re-framing 0.5589742

Several points have to be taught to the part-time R user if this code is to be maintained:

* The "!!" symbol does not have the same operator precedence as an assignment symbols such as "=" or ":=", so you must often place "!!"-expressions in extra parentheses.
* In any assignment we must use ":=" for assignment when using "!!" on the left-hand side of the assignment.

The above are just some syntax edge-cases, we haven’t even gone into teaching rlang::sym(), "!!", and the theory and semantics of quasi-quotation.

**seplyr solution**

[seplyr](https://winvector.github.io/seplyr/index.html) is an experiment to see what a referentially transparent (or completely value oriented) interface to dplyr would look like. Please don’t think of seplyr as an adapter (though it is, it sends all work to dplyr), but as an illustration of what a completely value-oriented dplyr might look like (i.e., one that did not capture un-evaluated user code through non-standard evaluation). Roughly seplyr is an experiment of the form: "what if one tried harder with something like the new dplyr::\*\_at() verbs."

Most of the seplyr methods are named \*\_se() and are designed to be very similar to their dplyr equivalents (and some are nearly identical to dplyr::\*\_at() methods, [rename\_se()](https://winvector.github.io/seplyr/reference/rename_se.html) being a notable exception).

library("seplyr")

suppressPackageStartupMessages(library("glue"))

d %>%

group\_by\_se(idCol) %>%

mutate\_se(probScoreCol :=

glue('exp({linkScoreCol} \* scale)/

sum(exp({linkScoreCol} \* scale))')) %>%

arrange\_se(c(probScoreCol, categoryCol)) %>%

mutate\_se(indicatorCol := "row\_number() == n()") %>%

filter\_se(indicatorCol) %>%

ungroup() %>%

select\_se(c(idCol, categoryCol, probScoreCol)) %>%

rename\_se(outcomeCol := categoryCol) %>%

arrange\_se(idCol)

## # A tibble: 2 x 3

## subjectID diagnosis probability

## <dbl> <chr> <dbl>

## 1 1 withdrawal behavior 0.6706221

## 2 2 positive re-framing 0.5589742

The concept is:

seplyr accepts general expressions many more places, but with proper organization and using a few temp-columns you really only *need* the full generality in mutate().

seplyr has its own issues:

* It also needs a "[:=](https://winvector.github.io/seplyr/reference/named_map_builder.html)" operator for assignment.
* It insists on multiple arguments coming in as vectors (hence the use of "c()" throughout).
* It runs into a bit of trouble with verbs that take expressions (mutate\_se() being the most complicated) in that it needs a helper to substitute in the name of the variable holding the column name, which is later substituted out for the actual column name by seplyr. In this example we used glue::glue() to perform the substitution, but we could also try paste0() or gsub().

The lesson from seplyr is the mutate() verb does indeed need some kind of expression manipulation tooling (direct string manipulation feeling too crude). However, for the rest of the verbs the value oriented notation is in fact quite natural, and really in no sense inferior to the dplyr originals.

The dplyr logistic scoring function example is as follows.

# scoring function wrapping code where the

# columns of interest were originally

# subjectID, surveyCategory, and assessmentTotal.

# function lets us re-map columns to new data later.

logistic\_score <- function(data, scale,

subjectID = "subjectID",

surveyCategory = "surveyCategory",

assessmentTotal = "assessmentTotal") {

let(mapsyms(subjectID, surveyCategory, assessmentTotal),

data %>%

group\_by(subjectID) %>%

mutate(probability =

exp(assessmentTotal \* scale)/

sum(exp(assessmentTotal \* scale))) %>%

arrange(probability, surveyCategory) %>%

mutate(isDiagnosis = row\_number() == n()) %>%

filter(isDiagnosis) %>%

ungroup() %>%

select(subjectID, surveyCategory, probability) %>%

rename(diagnosis = surveyCategory) %>%

arrange(subjectID)

)

}

# new data where columns of interest are

# student, surveyCategory, and points.

d <- data.frame(

student = c(1, 1, 2, 2),

surveyCategory = c(

'withdrawal behavior', 'positive re-framing',

'withdrawal behavior', 'positive re-framing'

),

points = c(5, 2, 3, 4),

stringsAsFactors = FALSE

)

# apply the function to new data

# re-specifying columns to the names we need

d %>%

logistic\_score(scale = 0.237,

subjectID = "student",

assessmentTotal = "points") %>%

knitr::kable()

| **student** | **diagnosis** | **probability** |
| --- | --- | --- |
| 1 | withdrawal behavior | 0.6706221 |
| 2 | positive re-framing | 0.5589742 |

The replace with values convention is particularly handy for converting one-off (or ad-hoc) analyses into re-usable functions by pasting code into a let-block() *without* additional alteration (when you can get away with that, as above). For harder tasks (converting code that isn’t suitable for the replace with values convention), we suggest the mixed case convention (which will now define).

**The let(X=x) (mixed case) convention**

For cases where the original code already has a mixture of parametric specifications (column names taken from variables) and non-parametric specifications (column names captured from un-evaluated code) I suggest using the "mixed case" convention. In mixed case convention all upper case symbols are used for replacement and lower case are taken as values. This is just a convention (the code does not implement the above as a rule) and we specify it by forming let alias maps of the form qc(X=x) which means in the let-block any instances of X are replaced with the name stored in x and (naturally) any instances of x are left alone.

Hers is a data.table function example using the mixed case convention.

logistic\_score <- function(data, scale,

subjectID = "subjectID",

surveyCategory = "surveyCategory",

assessmentTotal = "assessmentTotal") {

let(qc(SUBJECTID, SURVEYCATEGORY, ASSESSMENTTOTAL) :=

c(subjectID, surveyCategory, assessmentTotal),

{

dDT <- data.table::data.table(data)

setnames(dDT, surveyCategory, "diagnosis")

dDT[, expaTs := exp(ASSESSMENTTOTAL \* scale)]

# precalculate -> this uses gsum internally

dDT[, sum\_expaTs := sum(expaTs), SUBJECTID]

dDT[, probability := expaTs / sum\_expaTs]

dDT[, c(assessmentTotal, "expaTs", "sum\_expaTs") := NULL]

setorder(dDT, SUBJECTID,-probability, diagnosis)

dDT[, .SD[1], SUBJECTID]

})

}

d %>%

logistic\_score(.,

scale = 0.237,

subjectID = "student",

assessmentTotal = "points") %>%

knitr::kable(.)

| **student** | **diagnosis** | **probability** |
| --- | --- | --- |
| 1 | withdrawal behavior | 0.6706221 |
| 2 | positive re-framing | 0.5589742 |

The commonly suggested way to use symbolic column names with data.table (without wrapr) is to use quote()/as.symbol() and eval():

COLUMNNAME = as.symbol("x")

dt <- data.table::data.table(x = c(2, 3))

dt[, eval(COLUMNNAME) := eval(COLUMNNAME) + 1]

print(dt)

## x

## 1: 3

## 2: 4

Again, the wrapr equivalent is:

COLUMNNAME = "x"

dt <- data.table::data.table(x = c(2, 3))

let("COLUMNNAME" := COLUMNNAME,

dt[, COLUMNNAME := COLUMNNAME + 1]

)

print(dt)

## x

## 1: 3

## 2: 4

Obviously once you are dealing with both names and values (no matter what system you are using) you must take care in tracking which symbols refer to names and which symbols refer to values.

To use the wrapr mixed case convention:

* Build a name map using the qc(...) := c(..) notation. Reserve uppercase symbols for symbols to be replaced and lower case symbols for variable holding names of columns.
* Use the uppercase version of each symbol in non-standard contexts, and lower-case in standard contexts.

The mixed case convention is *very* powerful.

The 1.2.0 (currently development) version of wrapr adds a new function map\_upper() which allows writing the qc(SUBJECTID, SURVEYCATEGORY, ASSESSMENTTOTAL) := c(subjectID, surveyCategory, assessmentTotal) simply as map\_upper(subjectID, surveyCategory, assessmentTotal):

subjectID = "student"

surveyCategory = "surveyCategory"

assessmentTotal = "points"

map\_upper(subjectID, surveyCategory, assessmentTotal)

## $SUBJECTID

## [1] "student"

##

## $SURVEYCATEGORY

## [1] "surveyCategory"

##

## $ASSESSMENTTOTAL

## [1] "points"

And we use the uppercase/lowercase convention to mark what portions of code we wish to be substituted/re-written.

I would like to call out that all of these wrapr features (:=, qc(), mapsyms() map\_upper(), let()) are concrete functions that can be used separately or used together. That is: := isn’t a symbol that has a new interpretation only in let() blocks, it is a inline function that actually builds named vectors, and these named vectors in turn happen to be able to specify the mappings let() needs. This allows you to learn and test these functions separately (and allows you to find new uses for them in your own code). For example: if you find a new way to use let() blocks that needs a new mapping function, you can build that function (as the current functions are not wired into let(), so are not magic or privileged).

For multi-expression let()-blocks we must add {}. For := to work we must have wrapr‘s definition active, which we achieved by loading the wrapr package after loading the data.table package. data.table‘s use of := should continue to be correct as that is always performed by data.table itself, where wrapr‘s definition can not interfere.

**Additional q\*() methods**

wrapr supplies additional q\*() methods.

* qae() "quote assignment expression" where both sides of assignments is taken as un-evaluated. I.e.: qae(x = 5+1) yields c(‘x’ = ‘5 + 1’) regardless if x is bound or unbound in the environment. This is a bit of a complement to := which looks-up bindings/references (i.e.: x = "z"; x := 5+1 returns c(‘z’ = ‘6’)).
* qe() "quote expressions" for quoting complex expressions. Similar to quote(), except it returns a list of strings (not a language object). The qe() method is not as interesting to end-users as the other methods mentioned, it is designed to help in implementation of methods that take a non-assignment expression or list of expressions such as rquery::select\_rows\_nse().

**Take Away**

wrapr supplies some powerful and convenient R notations.

* := is a powerful convenience function.
* wrapr::qae(), wrapr::qc(), and wrapr::qe() can convert many "value oriented" (or standard evaluation) interfaces into "name capturing" (or non-standard evaluation) interfaces, making them slightly more concise (for an example, please see seplyr::mutate\_se()).
* wrapr::let() can convert many non-standard evaluation interfaces back into value oriented interfaces, making them easier to program over.