**Named Map Builder**

First I will demonstrate wrapr‘s  [named\_map\_builder: :=](https://winvector.github.io/wrapr/reference/named_map_builder.html). The named map builder adds names to vectors and lists by nice "names on the left and values on the right" notation.

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:

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"](http://www.win-vector.com/blog/2017/11/let-xx-in-r/).

The let() method of building functions works well with [dplyr](https://cran.r-project.org/package=dplyr) and [data.table](https://cran.r-project.org/package=data.table). For each of these let’s show code for the "by hand logistic scoring" example from ["Let’s Have Some Sympathy For The Part-time R User"](http://www.win-vector.com/blog/2017/08/lets-have-some-sympathy-for-the-part-time-r-user/).

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 ([from here](http://www.win-vector.com/blog/2018/01/base-r-can-be-fast/#comment-66751)) 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\_row\_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()](https://winvector.github.io/seplyr/reference/mutate_se.html)).
* wrapr::let() can convert many non-standard evaluation interfaces back into value oriented interfaces, making them easier to program over.