# Topic 5: Advance Loop Functions

ISOM3390: Business Programming in R

# Motivation

```
(df <- data.frame(a = rnorm(10), b = runif(10, 0, 20), c = rnorm(10, 4, 8)))
```

```
##              a         b          c
## 1  -0.35104690  6.119656  12.766427
## 2   0.79404524 14.536304   5.083920
## 3   1.75338445  8.026134 -12.850266
## 4  -0.86617239  4.763456   6.294404
## 5   0.25082320 14.371001 -12.877296
## 6  -0.06827350  9.002077 -11.097889
## 7   0.78997226  7.374773   8.432855
## 8   0.01145692  9.817640 -13.793480
## 9  -0.47044666  4.244544 -11.232877
## 10 -0.32724470 16.640394   7.294507
```

To compute, say, the median of each column, we could do with copy-and-paste three times:

```
mean(df$a)
```

```
## [1] 0.1516498
```

```
mean(df$b)
```

```
## [1] 9.489598
```

```
mean(df$c)
```

```
## [1] -2.19797
```

What if the data frame has many rows? a `for` loop may save our lives:

```
output <- vector("double", ncol(df))
for (i in seq_along(df)) {
    output[[i]] <- mean(df[[i]])
}
output
```

```
## [1]  0.1516498  9.4895980 -2.1979695
```

However, loops are slow and not very expressive.

Besides, writing loops is not particularly easy when working interactively with the console.

# Lessons Learned from Vector Arithmetic

```
v1 <- c(4, 6, 8, 24)
v2 <- c(34, 32.4, 12, 2.7)
v1 + v2
```

```
## [1] 38.0 38.4 20.0 26.7
```

Can we vectorize the `mean()` function to make it operate on whole objects as `+` does?

In R, there is a family of functions that give us the ability to apply functions to each element of a vector, a list, or an array.

With them, we can wrap up `for` loops in a function, and call that function instead of using the `for` loop directly.

# `lapply()`

`lapply()` (short for "list apply") takes a function, applies it to each element in a vector or a list (can also accept vector inputs), and returns the results in the form of a list.

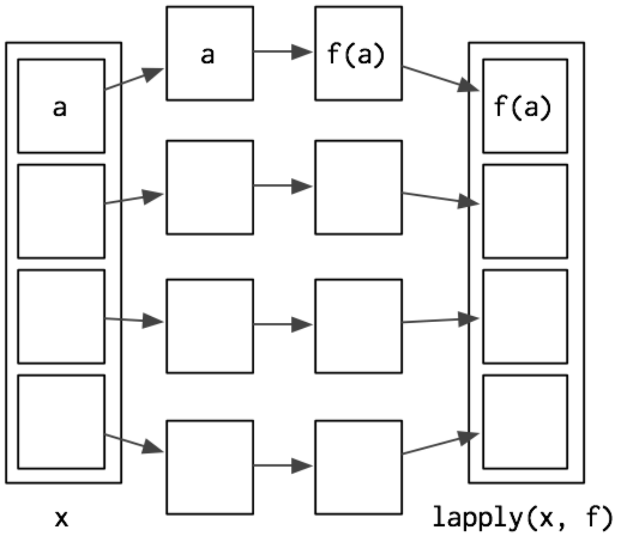
```
str(lapply)

## function (X, FUN, ...)

lapply(df, mean)

## $a
## [1] 0.1516498
##
## $b
## [1] 9.489598
##
## $c
## [1] -2.19797

# The names of the original list is preserved in the output.
```




When passing a function to another functin, we do not need to include the parentheses () as we do when calling a function

# Specifying Other Arguments for the Function to be Applied

The elements of the 1st argument to `lapply()` are always supplied as the 1st argument of the function we are applying.

```
x <- 1:3
str(runif)
```

random uniform function.

```
## function (n, min = 0, max = 1)
```

```
lapply(x, runif)
```

```
## [[1]]
## [1] 0.1168429
##
## [[2]]
## [1] 0.7549287 0.4454629
##
## [[3]]
## [1] 0.6983129 0.8288465 0.4325786
```

The `...` construct takes the remaining arguments and passes them down to the function being applied to the elements of the list.

```
lapply(x, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 6.56211
##
## [[2]]
## [1] 4.117143 2.885973
##
## [[3]]
## [1] 0.07429535 6.77194593 5.95763301
```

What if the elements of the list is passed down to the function as an argument other than the 1st one?

# Use of Anonymouse Functions

If we want to vary a different argument, we can use an **anonymous function**.

```
str(rep.int)
```

```
## function (x, times)
```

```
lapply(c(2, 3, 6), function(x) rep.int(3, times = x))
```

```
## [[1]]
## [1] 3 3
##
## [[2]]
## [1] 3 3 3
##
## [[3]]
## [1] 3 3 3 3 3 3
```

`lapply()` and other members in the `apply` family of functions make heavy use of anonymous functions.

# Applying Related Functions

Because functions are treated as **"first-class citizens"** in R, `lapply()` can work with groups of related functions as with lists of any other type of objects.

```
compute_mean <- list(base = function(x) mean(x),
                     sum = function(x) sum(x) / length(x),
                     manual = function(x) {
                        total <- 0
                        n <- length(x)
                        for (i in seq_along(x)) total <- total + x[i] / n
                        total
                        }
                     )

x <- runif(100)

unlist(lapply(compute_mean, function(f) f(x)))
```

```
##     base      sum   manual
## 0.499766 0.499766 0.499766
```

Define a list of functions. (For computing mean)

Pass this list of functions to lapply.

# Other Variations of `lapply()`

Other variations of `lapply()` simply use different types of input or output:

| | Descriptions |
|---|---|
| **sapply()** | produces a result of the simplest type possible, such as vectors, matrices, and lists, instead of lists only. |
| **mapply()** | applies a function in parallel over a set of arguments. |
| **apply()** | evaluates a function over the margins of an array. |
| **tapply()** | groups the elements of a vector and applies a function over the resulted subsets. |

# Vector Output: `sapply()`

`sapply()` (short for "simplified [l]apply") behaves similarly to `lapply()` except that it tries to simplify the results if possible.

Essentially, `sapply()` calls `lapply()` on its input and then applies the following simplifying algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

```
lapply(df, mean)
```

```
## $a
## [1] 0.1516498
##
## $b
## [1] 9.489598
##
## $c
## [1] -2.19797
```

```
sapply(df, mean)
```

```
##          a          b          c
##  0.1516498  9.4895980 -2.1979695
```

# Multiple Inputs: `mapply()`

`mapply()` (short for "multivariate apply") is a multivariate version of `sapply()` which applies a function in parallel over a set of arguments.

```
str(mapply)
```

```
## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

```
str(rnorm)
```

```
## function (n, mean = 0, sd = 1)
```

```
mapply(rnorm, 1:4, 1:4)
```

Pass multiple argument.

```
## [[1]]
## [1] 1.840933
##
## [[2]]
## [1] 2.389382 3.913044
##
## [[3]]
## [1] 2.595046 2.678158 2.955205
##
## [[4]]
## [1] 4.709207 3.013502 4.643779 3.765802
```

```
# The 3rd argument is re-cycled to the length of the longest
mapply(rnorm, 1:4, 1:4, 2)
```

```
## [[1]]
## [1] 2.582738
##
## [[2]]
## [1] 0.6602029 5.6928659
##
## [[3]]
## [1] -0.7596419  3.9734419  2.9928412
##
## [[4]]
## [1] 3.555540 4.953982 6.179296 2.539697
```

# Vectorizing a Function

In many statistical applications, we want to calculate the **sum of squares** $\Sigma_{i=1}^{n} \frac{(x_i - \mu)^2}{\sigma^2}$ to find the optimal $\mu$ and $\sigma$. Implement it with an R function:

```
sumsq <- function(mu, sigma, x) sum(((x - mu)/sigma)^2)
```

Generate some data to investigate its effect:

```
x <- rnorm(100)
sumsq(1, 1, x)
```

if sumsp <- function(a, b, c) a+b+c, then invoke sumsp(1, 1, c(1,2,3,4)), it will return a vector of (3,4,5,6)

```
## [1] 171.8538
```

Suppose we want to evaluate or plot it for 10 different choices of `mu` or `sigma`?

```
sumsq(1:10, 1:10, x)  # Does it print 10 values?
```

It will print 100 values because th last vector x has length 100.

Since x has size() of 100, so the function will be called 100 times. However, it will only return a single value as the sum of all 100 results.

Recycling rule can be applied here, unless all object length is a multiple of shorter object length.

When **vectorizing** a function, `mapply()` allows multiple arguments to the function to vary.

How about we changing the code to the following:

```
mapply(sumsq, 1:10, 1:10, x)  # Does it print 10 values?
```

It will call sumsq 100 times, because the length(x) is 100, then return a vector with size 100.

The summation of all these 100 itesm in this resulted vector is exactly equal to the single value generated in last page.

# `Vectorize()`

The `MoreArgs` argument of `mapply()` takes a list of arguments that will be supplied as constant inputs to each call.

```
mapply(sumsq, 1:10, 1:10, MoreArgs = list(x = x))
```

```
##  [1] 171.85377 108.94464  99.96703  97.72676  97.10211  96.98549  97.04903
##  [8]  97.17699  97.32410  97.47177
```

`Vectorize()` is a function wrapper for `mapply()` and provides more convenient interface for use.

```
vsumsq <- Vectorize(sumsq, c("mu", "sigma"))
vsumsq(1:10, 1:10, x)
```

```
##  [1] 171.85377 108.94464  99.96703  97.72676  97.10211  96.98549  97.04903
##  [8]  97.17699  97.32410  97.47177
```

What if the sizes for two parameters are different?

# Array Input: `apply()`

```
(m <- matrix(rnorm(12), 3, 4))
```

```
##           [,1]       [,2]       [,3]       [,4]
## [1,] -1.778508 -2.3192513 -1.2354000 -0.1813525
## [2,] -2.097801  0.6052652  1.3973194  0.5826304
## [3,] -1.706114 -0.3986137  0.3799567  0.7268335
```

```
sapply(m, mean)
```

```
##  [1] -1.7785080 -2.0978009 -1.7061142 -2.3192513  0.6052652 -0.3986137
##  [7] -1.2354000  1.3973194  0.3799567 -0.1813525  0.5826304  0.7268335
```

`lapply()` and `sapply()` treat the matrices and arrays as though they were vectors, whereas `apply()` (short for "array apply") evaluates a function over the **margins of an array**.

```
str(apply)
```

```
## function (X, MARGIN, FUN, ...)
```

```
apply(m, 2, mean)  ## column mean
```

Represents column

```
## [1] -1.8608077 -0.7041999  0.1806254  0.3760371
```

```
apply(m, 1, sum)  ## row sum
```

Represents row

```
## [1] -5.5145117  0.4874140 -0.9979376
```

The `MARGIN` argument essentially indicates which dimension of the array we want to preserve or retain.

# Col/Row Sums and Means

Shortcut functions perform column/row sums and means. They are much faster and more descriptive:

```
rowSums(m) <=> apply(m, 1, sum)

rowMeans(m) <=> apply(m, 1, mean)

colSums(m) <=> apply(m, 2, sum)

colMeans(m) <=> apply(m, 2, mean)
```

```
apply(m, 2, mean)  ## column mean

## [1] -1.8608077 -0.7041999  0.1806254  0.3760371

apply(m, 1, sum)  ## row sum

## [1] -5.5145117  0.4874140 -0.9979376
```

```
colMeans(m)  ## column mean

## [1] -1.8608077 -0.7041999  0.1806254  0.3760371

rowSums(m)  ## row sum

## [1] -5.5145117  0.4874140 -0.9979376
```

# Other Ways to Apply

- Pass the optional arguments to `FUN` via the `...` construct:

```
apply(m, 1, quantile, probs = 0.5)  # row median
```

```
## [1] -1.506953968  0.593947778 -0.009328489
```

- Preserve more than 1 dimension:

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
apply(a, c(1, 2), mean)
```

```
##            [,1]       [,2]
## [1,] -0.3825466 -0.3717463
## [2,]  0.1516039 -0.1135301
```

# A Common Problem

```
(frogger_scores <- data.frame(player = rep(c("Nick", "Charles", "Samuel"), times = c(4,
    3, 5)), score = round(rlnorm(12, 8), -1)))
```

```
##     player score
## 1     Nick  1090
## 2     Nick  4050
## 3     Nick  2920
## 4     Nick  2900
## 5  Charles  8570
## 6  Charles  1160
## 7  Charles  1630
## 8   Samuel  1090
## 9   Samuel  3010
## 10  Samuel  3880
## 11  Samuel 16250
## 12  Samuel  1820
```

How can we calculate some statistic on a variable (`score`) that has been split into groups (defined by `player`)?

# Splitting a Data Frame: `split()`

`split()` takes a vector or a data frame and splits it into groups determined by a factor or a list of factors:

```
str(split)
```

```
## function (x, f, drop = FALSE, ...)
```

First, split the datas by `player`:

```
(scores_by_player <- split(frogger_scores$score, frogger_scores$player))
```

```
## $Charles
## [1] 8570 1160 1630
##
## $Nick
## [1] 1090 4050 2920 2900
##
## $Samuel
## [1]  1090  3010  3880 16250  1820
```

# The Apply and Combine Steps

Next, apply the (`mean()`) function to each element:

```
(list_of_means_by_player <- lapply(scores_by_player, mean))
```

```
## $Charles
## [1] 3786.667
##
## $Nick
## [1] 2740
##
## $Samuel
## [1] 5210
```

Finally, combine the result into a single vector:

```
(mean_by_player <- unlist(list_of_means_by_player))
```

```
##  Charles     Nick   Samuel
## 3786.667 2740.000 5210.000
```

The **apply** and **combine** steps can be condensed into one by using `sapply()`.

Alternatively:

```
(frogger_by_player <- split(frogger_scores, frogger_scores$player))
```

```
## $Charles
##    player score
## 5 Charles  8570
## 6 Charles  1160
## 7 Charles  1630
##
## $Nick
##   player score
## 1   Nick  1090
## 2   Nick  4050
## 3   Nick  2920
## 4   Nick  2900
##
## $Samuel
##    player score
## 8  Samuel  1090
## 9  Samuel  3010
## 10 Samuel  3880
## 11 Samuel 16250
## 12 Samuel  1820
```

```
(sapply(frogger_by_player, function(x) mean(x$score)))
```

```
##  Charles     Nick   Samuel
## 3786.667 2740.000 5210.000
```

# A More Complex Problem: Enron Emails

```
## # A tibble: 150 x 4
##    time       from  to        n
##    <chr>      <fct> <fct> <int>
##  1 1999-01-04 114   65        2
##  2 1999-01-04 114   169       2
##  3 1999-01-07 114   110       4
##  4 1999-01-07 114   112       4
##  5 1999-01-07 114   169       2
##  6 1999-01-08 114   145       2
##  7 1999-01-08 114   169       8
##  8 1999-01-12 114   169       4
##  9 1999-01-13 114   22        2
## 10 1999-01-13 114   29        2
## # ... with 140 more rows
```

```
length(levels(small_corpus$from))
```

```
## [1] 11
```

```
length(levels(small_corpus$to))
```

```
## [1] 21
```

Suppose that we want to quantify the intensity of email exchange between a pair of employees during this period?

```
small_corpus_by_pair <- split(small_corpus$n, list(small_corpus$from, small_corpus$to))
str(small_corpus_by_pair, list.len = 15)
```

```
## List of 231
##  $ 22.6   : int(0)
##  $ 38.6   : int(0)
##  $ 50.6   : int(0)
##  $ 65.6   : int(0)
##  $ 107.6  : int(0)
##  $ 112.6  : int(0)
##  $ 114.6  : int(0)
##  $ 145.6  : int(0)
##  $ 155.6  : int(0)
##  $ 160.6  : int(0)
##  $ 169.6  : int 2
##  $ 22.11  : int(0)
##  $ 38.11  : int [1:5] 2 2 4 2 2
##  $ 50.11  : int(0)
##  $ 65.11  : int(0)
##   [list output truncated]
```

- With multiple factors and many levels, creating an interaction can result in many levels that are empty.

- We can drop empty levels when calling the `split()` function:

```
small_corpus_by_pair <- split(small_corpus$n, list(small_corpus$from, small_corpus$to),
    drop = TRUE)
str(small_corpus_by_pair, list.len = 15)
```

```
## List of 41
##  $ 169.6  : int 2
##  $ 38.11  : int [1:5] 2 2 4 2 2
##  $ 65.22  : int [1:2] 1 1
##  $ 114.22 : int [1:2] 2 2
##  $ 160.22 : int [1:3] 3 1 1
##  $ 114.29 : int 2
##  $ 65.38  : int 2
##  $ 114.38 : int [1:6] 2 2 2 2 4 2
##  $ 169.46 : int 2
##  $ 50.50  : int 2
##  $ 114.50 : int 2
##  $ 22.65  : int 2
##  $ 38.65  : int [1:2] 2 2
##  $ 112.65 : int 2
##  $ 114.65 : int [1:4] 2 2 4 2
##   [list output truncated]
```

# The "Split-Apply-Combine" Paradigm

The basic principle is as follows:

- **Split** a data set into (manageable) piece (e.g., according to some factor)
- **Apply** a function to each piece (e.g., `mean()`)
- **Combine** all the pieces into a single output (e.g., an array)

However, **"split-apply-combine"** is such a common data analysis paradigm for which we need something easier.

# Group Apply: `tapply()`

`tapply()` groups the elements of a vector and applies a function over the resulted subsets.

```
str(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

```
tapply(frogger_scores$score, frogger_scores$player, mean)
```

```
##  Charles     Nick   Samuel
## 3786.667 2740.000 5210.000
```

`tapply()` can be thought of as a combination of `split()` and `sapply()` for **vectors only**.

```
tapply2 <- function(x, group, f, ..., simplify = TRUE) {
    pieces <- split(x, group)
    sapply(pieces, f, ..., simplify = simplify)
}
tapply2(frogger_scores$score, frogger_scores$player, mean)
```

```
##  Charles     Nick   Samuel
## 3786.667 2740.000 5210.000
```

```r
tapply(small_corpus$n, list(small_corpus$from, small_corpus$to), sum)
```

```
##      6 11 22 29 38 46 50 65 96 107 110 112 114 123 128 145 155 160 165 167
## 22  NA NA NA NA NA NA NA  2 NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
## 38  NA 12 NA NA NA NA NA  4  2  NA  NA   2   2  NA  NA  NA  NA  NA  NA  NA
## 50  NA NA NA NA NA NA  2 NA NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  16
## 65  NA NA  2 NA  2 NA NA NA NA  NA  NA  NA  NA  NA  NA  NA  NA   2  NA  NA
## 107 NA NA NA NA NA NA NA NA NA  NA  NA  NA  14  NA  NA  NA  NA  NA  NA  NA
## 112 NA NA NA NA NA NA NA  2 NA   2  NA  NA   2  NA  NA  NA  NA  NA  NA  NA
## 114 NA NA  4  2 14 NA  2 10 NA  24  16  12  NA   2  NA   8  56   2  10  NA
## 145 NA NA NA NA NA NA NA NA NA  NA  NA  NA  NA  NA   2  NA  NA  NA  NA  NA
## 155 NA NA NA NA NA NA NA NA NA  NA  16  NA  16  NA  NA   2  NA  NA  NA  NA
## 160 NA NA  5 NA NA NA NA NA NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
## 169  2 NA NA NA NA  2 NA NA NA  NA  14  NA  50  NA  NA  NA   8  NA  14  NA
##     169
## 22   NA
## 38   NA
## 50   NA
## 65   NA
## 107  NA
## 112  NA
## 114  60
## 145  NA
## 155   4
## 160  NA
## 169  NA
```

# Higer-Order Functions and Functionals

A **higher-order function** is a function that does at least one of the following:

- returns a function as its result
  - closures, functions returned by another function
- takes one or more functions as arguments (i.e. procedural parameters)
  - **functionals**

> *"To become significantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. Messy code often hides bugs."*
>
> *— Bjarne Stroustrup*

Functionals implemented in base R are efficient and less error prone by better communicating intent.

# Why Use `apply` Functions Instead of `for` Loops?

- The code is cleaner (once we're familiar with the concept). The code can be easier to code and read, and less error prone because we don't have to deal with subsetting and saving the results.
- `apply` functions can be faster than `for` loops, sometimes dramatically.

# What's Wrong with `apply` Functions?

- Inconsistent syntax.
  - E.g., with `tapply()` and `sapply()`, the simplify argument is called `simplify`. With `mapply()`, it's called `SIMPLIFY`. With `apply()`, the argument is absent.
  - hard to remember
- Cover only a partial set of all possible combinations of input and output types
  - requires additional work for data transformation

| | List | Array | Data Frame |
|---:|---:|---:|---:|
| **List** | lapply | sapply | NA |
| **Array** | NA | apply | NA |
| **Function Arguments** | mapply | mapply | NA |

# A Quick Introduction to `plyr`

`plyr` allows us to smoothly apply the **split-apply-combine** strategy.

- It builds on the built-in `apply` functions and can be regarded as the generalization of `tapply()`.
- It provides a set of consistently named functions with consistently named arguments and gives us control over their input and output formats.
  - has a common syntax
  - requires less code since it takes care of the input and output format
  - can be run in parallel

# `plyr` Basics

The basic format of the `plyr` functions is two letters followed by `ply()`, with the 1st letter referring to the format in and the 2nd to the format out.

The three main letters are:

- `d` = **data frame**
- `a` = **array** (includes **matrices**)
- `l` = **list**

E.g., `ddply` means: take a data frame, split it up, do something to it, and return a data frame.

# `ddply()`

```
library(plyr)
str(ddply)
```

```
## function (.data, .variables, .fun = NULL, ..., .progress = "none",
##     .inform = FALSE, .drop = TRUE, .parallel = FALSE, .paropts = NULL)
```

- `.data`: data frame to process
- `.variables`: combination of variables to split by; support various specification syntax
  - Character: `c("from", "to")`
  - Numeric: `1:3`
  - Formula: `~ from + to`
- `.fun`: function to call on each piece
- `...`: extra arguments passed to `.fun`

# `summarise` for Group-Wise Summaries

```
summary_corpus <- ddply(small_corpus, c("from", "to"), summarise, total = sum(n),
    mean = mean(n))
head(summary_corpus, n = 15)
```

```
##    from  to total     mean
## 1    22  65     2 2.000000
## 2    38  11    12 2.400000
## 3    38  65     4 2.000000
## 4    38  96     2 2.000000
## 5    38 112     2 2.000000
## 6    38 114     2 2.000000
## 7    50  50     2 2.000000
## 8    50 167    16 2.666667
## 9    65  22     2 1.000000
## 10   65  38     2 2.000000
## 11   65 160     2 1.000000
## 12  107 114    14 3.500000
## 13  112  65     2 2.000000
## 14  112 107     2 2.000000
## 15  112 114     2 2.000000
```

`summarise` creates a new condensed data frame.

# `mutate` for Group-Wise Transformations

Unlike `summarise` that creates a new data frame, `mutate` modifies an existing data frame.

```
normalize_corpus <- ddply(small_corpus, c("from", "to"), mutate, mean = mean(n),
    sd = sd(n), normalized = (n - mean)/sd)
head(normalize_corpus, n = 15)
```

```
##          time from  to n     mean        sd normalized
## 1  1999-05-11   22  65 2 2.000000        NA         NA
## 2  1999-05-25   38  11 2 2.400000 0.8944272 -0.4472136
## 3  1999-05-27   38  11 2 2.400000 0.8944272 -0.4472136
## 4  1999-06-02   38  11 4 2.400000 0.8944272  1.7888544
## 5  1999-06-14   38  11 2 2.400000 0.8944272 -0.4472136
## 6  1999-06-15   38  11 2 2.400000 0.8944272 -0.4472136
## 7  1999-05-04   38  65 2 2.000000 0.0000000        NaN
## 8  1999-05-24   38  65 2 2.000000 0.0000000        NaN
## 9  1999-06-16   38  96 2 2.000000        NA         NA
## 10 1999-05-04   38 112 2 2.000000        NA         NA
## 11 1999-05-20   38 114 2 2.000000        NA         NA
## 12 1999-06-14   50  50 2 2.000000        NA         NA
## 13 1999-06-14   50 167 2 2.666667 1.6329932 -0.4082483
## 14 1999-06-15   50 167 2 2.666667 1.6329932 -0.4082483
## 15 1999-06-17   50 167 2 2.666667 1.6329932 -0.4082483
```

The transformations are executated iteratively so that later transformations can use the columns created by earlier ones.