Apply and friends

Marcus Wurzer

The `apply()` family of functions is often used to replace loops and can make computations much faster in many cases. This is mostly (but not always) true for `for` loops, whereas `while` loops are difficult to vectorize effectively. All the functions of the `apply()` family take a function as input and return a vector as output. This special type of function is called **functional**. Below, we make a distinction between functionals that operate on 1-dimensional (vectors, lists) and 2-dimensional data structures (matrices, data frames).

Manipulating vectors and lists

`lapply()`

We first generate a list of length 5. Each list element contains a vector of standard normally distributed numbers of different length:

```
set.seed(1)
l <- replicate(5, rnorm(sample(1:100, 1)))
l
```

```
## [[1]]
##  [1]  0.18364332 -0.83562861  1.59528080  0.32950777 -0.82046838  0.48742905
##  [7]  0.73832471  0.57578135 -0.30538839  1.51178117  0.38984324 -0.62124058
## [13] -2.21469989  1.12493092 -0.04493361 -0.01619026  0.94383621  0.82122120
## [19]  0.59390132  0.91897737  0.78213630  0.07456498 -1.98935170  0.61982575
## [25] -0.05612874 -0.15579551 -1.47075238 -0.47815006  0.41794156  1.35867955
## [31] -0.10278773  0.38767161 -0.05380504 -1.37705956 -0.41499456 -0.39428995
## [37] -0.05931340  1.10002537  0.76317575 -0.16452360 -0.25336168  0.69696338
## [43]  0.55666320 -0.68875569 -0.70749516  0.36458196  0.76853292 -0.11234621
## [49]  0.88110773  0.39810588 -0.61202639  0.34111969 -1.12936310  1.43302370
## [55]  1.98039990 -0.36722148 -1.04413463  0.56971963 -0.13505460  2.40161776
## [61] -0.03924000  0.68973936  0.02800216 -0.74327321  0.18879230 -1.80495863
## [67]  1.46555486  0.15325334
##
## [[2]]
##  [1]  0.01915639  0.25733838 -0.64901008 -0.11916876  0.66413570  1.10096910
##  [7]  0.14377148 -0.11775360 -0.91206837 -1.43758624 -0.79708953  1.25408311
## [13]  0.77214219 -0.21951563 -0.42481028 -0.41898010  0.99698686 -0.27577803
## [19]  1.25601882  0.64667439  1.29931230 -0.87326211  0.00837096 -0.88087172
## [25]  0.59625902  0.11971764 -0.28217388  1.45598840  0.22901959  0.99654393
## [31]  0.78185918 -0.77677662 -0.61598991  0.04658030 -1.13038578  0.57671878
## [37] -1.28074943  1.62544730 -0.50069660  1.67829721 -0.41251989 -0.97228684
## [43]  0.02538287  0.02747534 -1.68018272  1.05375086 -1.11959910  0.33561721
##
## [[3]]
##  [1]  0.49418833 -0.17733048 -0.50595746  1.34303883 -0.21457941 -0.17955653
##  [7] -0.10019074  0.71266631 -0.07356440 -0.03763417 -0.68166048 -0.32427027
##
```

```
## [[4]]
##  [1] -0.47569828  0.79791644 -0.97400256  0.68937270 -0.95583910 -1.23170706
##  [7] -0.95689188 -0.86978287 -0.91068068  0.74127631  0.06851153 -0.32375075
## [13] -1.08650305 -1.01592895 -0.76779018 -1.11972006 -0.44817424  0.47173637
## [19] -1.18049068  1.47025700 -1.31142059 -0.09652492  2.36971991  0.89062648
## [25] -0.25218316 -0.86576375  0.58258600 -0.01252935 -0.37485476  0.31788574
## [31] -0.48880563  2.65865803  1.68027820  0.77958401  0.71324052 -0.54288194
##
## [[5]]
##  [1] -0.34859468 -1.00805458  1.88318254 -0.92897108 -0.29419645 -0.61495027
##  [7] -0.94707579  0.59897515 -1.52361488 -0.20618900 -0.57429541 -1.39016604
## [13] -0.07041738 -0.43087953 -0.59222537  0.98111616  0.53240936 -0.09045612
## [19]  0.15649049 -0.73731169 -0.20134121  1.10217660 -0.01674826  0.16178863
## [25]  2.02476139 -0.70369425  0.96079238  1.79048505 -1.06416516  0.01763655
## [31] -0.38990863 -0.49083275 -1.04571765 -0.89621126  1.26938716  0.59384095
## [37]  0.77563432  1.55737038 -0.36540180  0.81655645 -0.06063478 -0.50137832
## [43]  0.92606273  0.03693769 -1.06620017 -0.23845635  1.49522344  1.17215855
## [49] -1.45770721  0.09505623  0.84766496 -1.62436453  1.40856336 -0.54176036
## [55]  0.27866472 -0.19397274  1.57615818 -1.47554764 -0.14460821 -0.95320315
## [61]  0.40654273  2.22926220 -1.51449701 -0.06170742 -0.14727079
```

It would be cumbersome to compute the length of each list element separately (`length(l[[1]])` etc.), so we could go for a `for` loop:

```
out <- vector("list", length(l)) # empty list of length 5 to save the results
for (i in seq_along(l)) {
  out[[i]] <- length(l[[i]])
}
out
```

```
## [[1]]
## [1] 68
##
## [[2]]
## [1] 48
##
## [[3]]
## [1] 12
##
## [[4]]
## [1] 36
##
## [[5]]
## [1] 65
```

Since we only get single numbers (i.e., vectors of length one), we use the `unlist()` command to get the output in the more convenient vector form:

```
unlist(out)
```

```
## [1] 68 48 12 36 65
```

`lapply()` makes this task much simpler:

```
lapply(X = l, FUN = length)
```

```
## [[1]]
## [1] 68
```

```
##
## [[2]]
## [1] 48
##
## [[3]]
## [1] 12
##
## [[4]]
## [1] 36
##
## [[5]]
## [1] 65
```

In written form, this could be formulated as *Apply function `length` on each element of object `l`*. Of course, we could also use `unlist()` and leave out the argument names:

```
unlist(lapply(l, length))
```

```
## [1] 68 48 12 36 65
```

The results can also be of length > 1:

```
lapply(l, range)
```

```
## [[1]]
## [1] -2.214700  2.401618
##
## [[2]]
## [1] -1.680183  1.678297
##
## [[3]]
## [1] -0.6816605  1.3430388
##
## [[4]]
## [1] -1.311421  2.658658
##
## [[5]]
## [1] -1.624365  2.229262
```

We get the minimum and the maximum of each list element. What happens if we unlist the result?

```
unlist(lapply(l, range))
```

```
##  [1] -2.2146999  2.4016178 -1.6801827  1.6782972 -0.6816605  1.3430388
##  [7] -1.3114206  2.6586580 -1.6243645  2.2292622
```

We get a vector of length 10, i.e, we would have to construct a matrix to get a more meaningful representation:

```
matrix(unlist(lapply(l, range)), nrow = 2)
```

```
##           [,1]      [,2]       [,3]      [,4]      [,5]
## [1,] -2.214700 -1.680183 -0.6816605 -1.311421 -1.624365
## [2,]  2.401618  1.678297  1.3430388  2.658658  2.229262
```

But this isn't really necessary if we use one of the friends of `lapply()` (see below).

We may not only use named functions, but also functions defined within the scope of `lapply()`:

```
unlist(lapply(l, function(x) mean(x) ^ 2))
```

```
## [1] 0.0192023147 0.0018604163 0.0004520922 0.0031805691 0.0001448025
```

We already discussed that data frames are also lists. Consequently, `lapply()` can also be used when operating on data frames. We will use data set `mtcars` that is contained in R:

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

We can see that this is a data set providing some technical information about 32 cars. If we want to know the the storage mode of all the variables included in `mtcars`, we could use the following code:

```
unlist(lapply(mtcars, typeof))
```

```
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
## "double" "double" "double" "double" "double" "double" "double" "double"
##       am     gear     carb
## "double" "double" "double"
```

sapply() and vapply()

These two functionals are variants of `lapply()` that do not produce lists, but vectors, matrices, and arrays as output. The two functions differ in that `sapply()` tries to guess the output type while we directly have to enter it when using `vapply()`. This difference is important when things go wrong (i.e., errors occur): `sapply()` is simpler and therefore usually the better choice for interactive use, `vapply()` is more verbose and preferred for the usage within functions (because of the more informative error messages).

Above, we used `unlist(lapply(l, length))` to get the length of the list elements in vector form. We may use `sapply()` to get the same result without having to use `unlist()`:

```
sapply(l, length)
```

```
## [1] 68 48 12 36 65
```

Using `vapply()`, we have to provide the information that each element is a numeric vector of length 1, otherwise we get an error:

```
vapply(l, length)
```

```
## Error in vapply(l, length): argument "FUN.VALUE" is missing, with no default
```

```
vapply(l, length, numeric(1))
```

```
## [1] 68 48 12 36 65
```

Above we saw what happens when the results are not of length 1 (for `range()`). Using `sapply()`, we directly get the results in the matrix form we want:

```
sapply(l, range)
```

```
##           [,1]      [,2]       [,3]      [,4]      [,5]
## [1,] -2.214700 -1.680183 -0.6816605 -1.311421 -1.624365
## [2,]  2.401618  1.678297  1.3430388  2.658658  2.229262
```

And for `vapply`:

```r
vapply(l, range, numeric(2))
```

```
##             [,1]      [,2]       [,3]      [,4]      [,5]
## [1,] -2.214700 -1.680183 -0.6816605 -1.311421 -1.624365
## [2,]  2.401618  1.678297  1.3430388  2.658658  2.229262
```

We get an informative error message when misspecifying the "FUN.VALUE" argument:

```r
vapply(l, range, numeric(1))
```

```
## Error in vapply(l, range, numeric(1)): values must be length 1,
##  but FUN(X[[1]]) result is length 2
```

mapply() (and Map)

mapply is used when there are several inputs that vary, e.g., when we want to compute a weighted mean - a common task in descriptive statistics - using two lists, where list l contains the data vectors, and list k contains some Poisson-distributed weights (with the same list element lengths):

```r
l.len <- sapply(l, length)
k <- lapply(l.len, function(x) rpois(x, 5) + 1)
```

It is no problem to compute the mean of all the elements of l:

```r
sapply(l, mean)
```

```
## [1]  0.13857242  0.04313254  0.02126246 -0.05639653  0.01203339
```

When computing the weighted means (each observation in l is weighted by the corresponding weights given in k before computing the mean of a list element), we use mapply() as follows:

```r
mapply(weighted.mean, l, k)
```

```
## [1]  0.11980378 -0.04357112  0.12972000 -0.12667319 -0.01032071
```

Note that the order of function arguments is different (compared to lapply(), sapply(), and (vapply)). Map() is similar to mapply(), but doesn't try to reduce the results to a vector, matrix, or array (comparable to the difference between sapply() and lapply()):

```r
Map(weighted.mean, l, k)
```

```
## [[1]]
## [1] 0.1198038
## 
## [[2]]
## [1] -0.04357112
## 
## [[3]]
## [1] 0.12972
## 
## [[4]]
## [1] -0.1266732
## 
## [[5]]
## [1] -0.01032071
```

```r
unlist(Map(weighted.mean, l, k))
```

```
## [1]  0.11980378 -0.04357112  0.12972000 -0.12667319 -0.01032071
```

Manipulation of matrices and data frames

Here we deal with 2-dimensional structures, but the functionals presented subsequently work with higher-dimensional structures as well.

`apply()`

`apply()` is comparable to `sapply()`, but needs additional information about the dimensions to summarize over. For 2-dimensional structures, 1 stands for rows, and 2 stands for columns:

```r
X <- matrix(1:15, nrow = 5)
apply(X = X, MARGIN = 1, FUN = mean) # rowwise means
```

```
## [1]  6  7  8  9 10
```

```r
apply(X, 2, mean) # columnwise means
```

```
## [1]  3  8 13
```

`sweep()` and `outer()`

`sweep()` is often used in combination with `apply()` to "sweep" out the values of some summary statistic, e.g., the mean:

```r
sweep(X, 2, apply(X, 2, mean)) # subtract the column means c(1, 8, 13) from each row
```

```
##      [,1] [,2] [,3]
## [1,]   -2   -2   -2
## [2,]   -1   -1   -1
## [3,]    0    0    0
## [4,]    1    1    1
## [5,]    2    2    2
```

```r
sweep(X, 1, apply(X, 1, mean)) # subtract the row means c(6, 7, 8, 9, 10) from each column
```

```
##      [,1] [,2] [,3]
## [1,]   -5    0    5
## [2,]   -5    0    5
## [3,]   -5    0    5
## [4,]   -5    0    5
## [5,]   -5    0    5
```

On common task is scaling the rows of a matrix (i.e., the resulting values all lie between 0 and 1):

```r
(X1 <- sweep(X, 1, apply(X, 1, min), "-")) # subtract ("-") the rowwise min
```

```
##      [,1] [,2] [,3]
## [1,]    0    5   10
## [2,]    0    5   10
## [3,]    0    5   10
## [4,]    0    5   10
## [5,]    0    5   10
```

```r
(X2 <- sweep(X1, 1, apply(X1, 1, max), "/")) # divide by ("/") the rowwise max
```

```
##      [,1] [,2] [,3]
## [1,]    0  0.5    1
## [2,]    0  0.5    1
## [3,]    0  0.5    1
## [4,]    0  0.5    1
## [5,]    0  0.5    1
```

The same principle applies to the standardization (subtract mean, divide by sd):

```
m <- apply(X, 2, mean)
s <- apply(X, 2, sd)
sweep(sweep(X, 2, m), 2, s, "/")
```

```
##            [,1]       [,2]       [,3]
## [1,] -1.2649111 -1.2649111 -1.2649111
## [2,] -0.6324555 -0.6324555 -0.6324555
## [3,]  0.0000000  0.0000000  0.0000000
## [4,]  0.6324555  0.6324555  0.6324555
## [5,]  1.2649111  1.2649111  1.2649111
```

Compare:

```
scale(X)
```

```
##            [,1]       [,2]       [,3]
## [1,] -1.2649111 -1.2649111 -1.2649111
## [2,] -0.6324555 -0.6324555 -0.6324555
## [3,]  0.0000000  0.0000000  0.0000000
## [4,]  0.6324555  0.6324555  0.6324555
## [5,]  1.2649111  1.2649111  1.2649111
## attr(,"scaled:center")
## [1]  3  8 13
## attr(,"scaled:scale")
## [1] 1.581139 1.581139 1.581139
```

`outer` is used to create a matrix or array ouput taking multiple vectors as inputs. The input function is run over every combination of the inputs:

```
x <- 1:3
y <- 1:10
outer(x, y, "*")
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    3    4    5    6    7    8    9    10
## [2,]    2    4    6    8   10   12   14   16   18    20
## [3,]    3    6    9   12   15   18   21   24   27    30
```

Every value of `x` was multiplied with every value of `y`. An application of `outer()` is the computation of expected values in frequency tables.

Using `apply()` on data frames is straightforward:

```
apply(mtcars, 2, typeof)
```

```
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
## "double" "double" "double" "double" "double" "double" "double" "double"
##       am     gear     carb
## "double" "double" "double"
```

But be careful if there are variables of different modes (which is usually the case in data frames):

```
Y <- mtcars
apply(Y, 1, mean)
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive
##            29.90727            29.98136            23.59818            38.73955
##   Hornet Sportabout             Valiant          Duster 360           Merc 240D
##            53.66455            35.04909            59.72000            24.63455
##            Merc 230            Merc 280           Merc 280C          Merc 450SE
##            27.23364            31.86000            31.78727            46.43091
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental
##            46.50000            46.35000            66.23273            66.05855
##   Chrysler Imperial            Fiat 128         Honda Civic      Toyota Corolla
##            65.97227            19.44091            17.74227            18.81409
##       Toyota Corona    Dodge Challenger         AMC Javelin          Camaro Z28
##            24.88864            47.24091            46.00773            58.75273
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa
##            57.37955            18.92864            24.77909            24.88027
##      Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
##            60.97182            34.50818            63.15545            26.26273
```

```
apply(Y, 2, mean)
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

```
Y$am <- factor(Y$am, labels = c("automatic", "manual"))
apply(Y, 1, mean)
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive
##                  NA                  NA                  NA                  NA
##   Hornet Sportabout             Valiant          Duster 360           Merc 240D
##                  NA                  NA                  NA                  NA
##            Merc 230            Merc 280           Merc 280C          Merc 450SE
##                  NA                  NA                  NA                  NA
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental
##                  NA                  NA                  NA                  NA
##   Chrysler Imperial            Fiat 128         Honda Civic      Toyota Corolla
##                  NA                  NA                  NA                  NA
##       Toyota Corona    Dodge Challenger         AMC Javelin          Camaro Z28
##                  NA                  NA                  NA                  NA
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa
##                  NA                  NA                  NA                  NA
##      Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
##                  NA                  NA                  NA                  NA
```

```
apply(Y, 2, mean)
```

```
##  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
##   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
```

`tapply()` (Group apply)

`tapply()` is a generalization of `apply()` and often used to summarize a data set by different groups. We return to the `Y` data frame created from the `mtcars` data set. Suppose we want to summarize horsepower (Variable `hp`) separately for the two transmission categories (Variable `am`):

```
attach(Y)
tapply(hp, am, mean)
```

```
## automatic    manual
##  160.2632  126.8462
```

```
tapply(hp, am, sd)
```

```
## automatic    manual
##  53.90820  84.06232
```

```
tapply(hp, am, max)
```

```
## automatic    manual
##       245       335
```

```
tapply(hp, am, summary)
```

```
## $automatic
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    62.0   116.5   175.0   160.3   192.5   245.0
##
## $manual
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    52.0    66.0   109.0   126.8   113.0   335.0
```

```
detach(Y)
```

`tapply()` allows for ragged arrays, i.e., arrays where each row can have a different number of columns. We can see this in detail if we count the absolute frequencies of the factor levels of `am`:

```
table(Y$am)
```

```
##
## automatic    manual
##        19        13
```

We can see that there are 19 cars with automatic, but only 13 with manual transmission. `tapply()` first takes the variable of interest (`hp`), splits it by the values of the grouping variable (`am`), and then separately applies the function (`mean`, `sd`, etc.) to each of the resulting vectors. This also works if we have more than one grouping variable. Let's coerce variable `vs` (engine type) to a factor, too:

```
Y$vs <- factor(Y$vs, labels = c("V-shaped", "straight"))
attach(Y)
tapply(hp, list(am, vs), mean)
```

```
##           V-shaped  straight
## automatic 194.1667 102.14286
## manual    180.8333  80.57143
```

```
detach(Y)
```

Instead of attaching and detaching, we can also use the `with()` function (a sort of temporary attachment):

```
with(Y, tapply(hp, list(am, vs), mean))
```

```
##           V-shaped  straight
## automatic 194.1667 102.14286
## manual    180.8333  80.57143
```