data frames in R

Intuitively, data frames are similar to matrices; having two-dimensional rows-and-columns structure. The difference exists where each column in a list can be of different classes (modes). For example, one column can be numerical values, while another contains character strings. Therefore, just as lists are heterogeneous analogs of vectors in one-dimension, data frames are heterogeneous analogs of matrices for two-dimensional data.

Concretely, a data frame is a list, with the components of that list being equal0length vectors. R does not allow components to be other types of objects, including other data frames. Therefore, the result is a heterogeneous-data analogs of arrays, not common in practice. It will thus be assumed that all components of a data frame are vectors.

creating data frames in R

When building data frames in R, the default selection sets **stringsAsFactors = TRUE**, converting character vectors into ***factors*** (discussed later).

```
> kids <- c("Jack", "Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages, stringsAsFactors = FALSE)
> d
  kids ages
1 Jack   12
2 Jill   10
```

accessing data frames in R

Considering that **d** is a list, it can be accessed via component index values or component names:

```
> d[[1]]                  #accessing dataframe d with components indices
[1] "Jack" "Jill"

> d$kids                  #accessing dataframe d with component names
[1] "Jack" "Jill"
```

Additionally, data frame **d** can be treated as a matrix to view components (also illustrated in its structure):

```
> d[,1]                                 #print column one of dataframe d
[1] "Jack" "Jill"

> str(d)                                #examine matrix-like structure of dataframe d
'data.frame':  2 obs. of  2 variables:
 $ kids: chr  "Jack"
```

The structure of dataframe **d** illustrates two observations—two rows—that stores the data on two variables—two columns. It is safer and more clear to extra data from matrices using name references of the components with the **$** operator. However, in the construction of general code (packages and functions), it is necessary to use matrix-like notation **[ ]** and is also useful for extracting sub dataframes.

extracting sub data frames in R

Matrix operations also apply to data frames where appropriate. Most noted is often **filtering** of dataframes to extract various sub dataframes relevant to analysis.

```
> exams                        #print dataframe exams
  Exam 1 Exam 2 Quiz
1    2.0    3.3  4.0
2    3.3    2.0  3.7
3    4.0    4.0  4.0
4    2.3    0.0  3.3
5    2.3    1.0  3.3
6    3.3    3.7  4.0

> exams[2:5,]                  #subset rows 2 through 5 of dataframe exams
  Exam 1 Exam 2 Quiz
2    3.3      2  3.7
3    4.0      4  4.0
4    2.3      0  3.3
5    2.3      1  3.3

> exams[2:5,2]               #subset rows 2-5 of column 2 in exams (returned as vector)
[1] 2 4 0 1

> class(exams[2:5,2])        #print class of dataframe exams subset above
[1] "numeric"

> exams[2:5,2,drop = FALSE]    #specify the drop argument to maintain matrix integrity
  Exam 2
2      2
3      4
4      0
5      1

> class(exams[2:5,2,drop = FALSE])    #print the class of the subset with drop = FALSE
[1] "data.frame"
```

Additionally, dataframes can be filtered through sub dataframe extraction and specification of constraints.

```
> exams[exams$`Exam 1` >= 2.5,]
  Exam 1 Exam 2 Quiz
2    3.3    2.0  3.7
3    4.0    4.0  4.0
5    6    3.3    3.7  4.0
```

data frames with NA values Y in R

Assuming that NA values exist within the rows/columns of a dataframe, certain operations cannot be properly performed. In these cases, R should be specifically assigned to remove NA values from operations.

```
x <- c(2, NA, 4)
> mean(x)
[1] NA

> mean(x, na.rm = TRUE)
[1] 3
```

When dealing with NA values in larger datasets, it might be necessary to remove all observations where NA values are present. This is achieved through the **complete.cases( )** function as illustrated on the proceeding page:

```
> exams                  #print the exams dataframe with NA value in last row
  Exam 1 Exam 2 Quiz
1    2.0    3.3  4.0
2    3.3    2.0  3.7
3    4.0    4.0  4.0
4    2.3    0.0  3.3
5    2.3    1.0  3.3
6    3.3    3.7  4.0
7    2.0     NA  4.0

> mean(exams)           #attempt to return the mean of the exam dataframe
[1] NA
Warning message:
In mean.default(exams) : argument is not numeric or logical: returning NA

> complete.cases(exams)        #print the result of rows that exclude NA values
[1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

> cmplExams <- exams [complete.cases(exams),]  #subset the dataframe from non-NA obs.
> cmplExams
  Exam 1 Exam 2 Quiz
1    2.0    3.3  4.0
2    3.3    2.0  3.7
3    4.0    4.0  4.0
4    2.3    0.0  3.3
5    2.3    1.0  3.3
6    3.3    3.7  4.0
```

rbind( ) and cbind( ) with data frames in R

The **rbind( )** and **cbind( )** functions apply to dataframes equally, given compatible dataframe sizes; additional columns and rows are required to have the same length as the target dataframe. It is typical where an added row is in the form of another dataframe or list:

```
> d
  kids ages
1 Jack   12
2 Jill   10

> rbind(d, list("laura", 19))
   kids ages
1  Jack   12
2  Jill   10
3 laura   19
```

Additionally, new columns can be created from existing columns:

```
> eq <- cbind(exams, exams[,2] - exams[,1])
> eq
  Exam 1 Exam 2 Quiz exams[, 2] - exams[, 1]
1    2.0    3.3  4.0                    1.3
2    3.3    2.0  3.7                   -1.3
3    4.0    4.0  4.0                    0.0
4    2.3    0.0  3.3                   -2.3
5    2.3    1.0  3.3                   -1.3
6    3.3    3.7  4.0                    0.4
```

To avoid extraneous names being assigned to calculated columns, variables can be specified with the **names( )** function. Alternatively, the properties of lists (equal-length vectors) can be applied and calculated in a vectorized faction; the names can be assigned in the midst of this optimization (proceeding page):

```
> exams$ExamDiff <- exams$`Exam 2` - exams$`Exam 1`
> exams
  Exam 1 Exam 2 Quiz ExamDiff
1    2.0    3.3  4.0      1.3
2    3.3    2.0  3.7     -1.3
3    4.0    4.0  4.0      0.0
4    2.3    0.0  3.3     -2.3
5    2.3    1.0  3.3     -1.3
6    3.3    3.7  4.0      0.4
```

The above illustrates the addition of components to an existing list. The property of **recycling** can equally be exploited in the creation of components in existing lists.

```
> d
  kids ages
1 Jack   12
2 Jill   10

> d$one <- 1
> d
  kids ages one
1 Jack   12   1
2 Jill   10   1
```

apply( ) with data frames in R

The **apply( )** function can be used against dataframes, given that the columns are of the same type.

```
> apply(exams,1,max)                #finding the maximum grade each student received
[1] 4.0 3.7 4.0 3.3 3.3 4.0
```

merging data frames in R

In relational database theory, joining tables according to common variable values is an essential task. In R, the latter is accomplished through the **merge( )** function.

```
> merge(x, y)                          #the simplest form of a join in R
```

The above illustration assumes that the two data frames have one or more columns with names in common:

```
> d1
     kids states
1    Jack     CA
2    Jill     MA
3 Jillian     MA
4    John     HI

> d2
  ages    kids
1   10    Jill
2    7 Lillian
3   12    Jack

> d <- merge(d1, d2)  #merging d1 and d2 only returns records with common properties
> d
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
```

Utilized the **by.x** and **by.y** arguments in the **merge( )** function handles cases where variables contain the same information for joining, but lack consistency across naming conventions (illustrated on proceeding page):

```
> d3
  ages    pals
1   12    Jack
2   10    Jill
3    7 Lillian

> merge(d1, d3, by.x = "kids", by.y = "pals")
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
```

It is possible that duplicate matches will appear with the full results of a **merge( )** application.

```
> d1
      kids states
1    Jack     CA
2    Jill     MA
3 Jillian     MA
4    John     HI

> d2a <- rbind(d2, list(15, "Jill"))
> d2a
  ages    kids
1   10    Jill
2    7 Lillian
3   12    Jack
4   15    Jill

> merge(d1, d2a)     #One of the Jill's has unknown residence; erroneously assigned MA
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
3 Jill     MA   15
```

The above illustrations displays the unintended consequences of applying certain **merge( )** functions to dataframes in R. It is imperative to be aware of the data and how it is being applied in R functions.

applying functions to data frames $f(\boxplus)$ in R

Similar to lists, the **lapply( )** and **sapply( )** functions can be applied to dataframes in R; remembering that dataframes are special cases of lists (list components consisting of the dataframe's columns). Therefore, calling **lapply( )** on a dataframe with specified function **f( )**, will be called on each of the frame's columns, with the returned valued populated in a corresponding list.

```
> d
  kids ages
1 Jack   12
2 Jill   10

> dl <- lapply(d, sort) #create list dl with two vectors, sorted kids and sorted ages
> dl
$kids
[1] Jack Jill
$ages
[1] 10 12
```

Noting that **dl** is a list; dataframe coercion to a is accomplished through the **as.data.frame( )** function:

```
> as.data.frame(dl) #coercion to a dataframe has lost column references (Jack ≠ 10)
  kids ages
1 Jack   10
2 Jill   12
```