R Cookbook

by Paul Teetor

S Kwon, Department of Applied Statistics, Konkuk University

shkwon0522@konkuk.ac.kr

5. Data Structures

Vectors

- *Vector elements can have names.*

  Vectors have a names property, the same length as the vector itself, that gives names to the elements

```
> v <- c(10, 20, 30)
> names(v) <- c("Moe", "Larry", "Curly")
> print(v)
  Moe Larry Curly
   10    20    30
```

- *If vector elements have names then you can select them by name*

  Continuing the previous example:

```
> v["Larry"]
Larry
   20
```

5. Data Structures

Lists

- *Lists can be indexed by position*

  So lst[[2]] refers to the second element of lst. Note the double square brackets.

- *Lists let you extract sublists*

  So lst[c(2,3)] is a sublist of lst that consists of the second and third elements. Note the single square brackets.

- *List elements can have names*

  Both lst[["Moe"]] and lst$Moe refer to the element named Moe.

5. Data Structures

Mode: Physical Type

- In R, every object has a mode, which indicates how it is stored in memory:

Object	Example	Mode
Number	`3.1415`	numeric
Vector of numbers	`c(2.7.182, 3.1415)`	numeric
Character string	`"Moe"`	character
Vector of character strings	`c("Moe", "Larry", "Curly")`	character
Factor	`factor(c("NY", "CA", "IL"))`	numeric
List	`list("Moe", "Larry", "Curly")`	list
Data frame	`data.frame(x=1:3, y=c("NY", "CA", "IL"))`	list
Function	`print`	function

5. Data Structures

Mode: Physical Type

- The mode function gives us this information.

```
> mode(3.1415)                        # Mode of a number
[1] "numeric"
> mode(c(2.7182, 3.1415))             # Mode of a vector of numbers
[1] "numeric"
> mode("Moe")                         # Mode of a character string
[1] "character"
> mode(list("Moe","Larry","Curly")) # Mode of a list
[1] "list"
```

5. Data Structures

Scalars

- In R, a scalar is simply a vector that contains exactly one element.

```
> pi
[1] 3.141593

> length(pi)
[1] 1

> pi[1]
[1] 3.141593

> pi[2]
[1] NA
```

5. Data Structures

Matrices

- In R, a matrix is just a vector that has dimensions.
- A vector has an attribute called dim, which is initially NULL, as shown here.

```
> A <- 1:6
> dim(A)
NULL
> print(A)
[1] 1 2 3 4 5 6
```

- Watch what happens when we set our vector dimensions to 2 3 and print it.

```
> dim(A) <- c(2,3)
> print(A)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

5. Data Structures

Matrices

- A matrix can be created from a list, too.
- Like a vector, a list has a dim attribute, which is initially NULL.

```
> B <- list(1,2,3,4,5,6)
> dim(B)
NULL
```

- If we set the dim attribute, it gives the list a shape.

```
> dim(B) <- c(2,3)
> print(B)
     [,1] [,2] [,3]
[1,] 1    3    5
[2,] 2    4    6
```

5. Data Structures

Arrays

- The following example creates a 3-dimensional array with dimensions 2 3 2.

```
> D <- 1:12
> dim(D) <- c(2,3,2)
> print(D)
, , 1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
, , 2
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

5. Data Structures

Arrays

- This code snippet creates a matrix that is a mix of numeric and character data.

```
> C <- list(1, 2, 3, "X", "Y", "Z")
> dim(C) <- c(2,3)
> print(C)
     [,1] [,2] [,3]
[1,] 1    3    "Y"
[2,] 2    "X"  "Z"
```

5. Data Structures

Data Frames

- A data frame is a tabular (rectangular) data structure, which means that it has rows and columns.
- It is not implemented by a matrix, however. Rather, a data frame is a list.
- Because a data frame is both a list and a rectangular structure, R provides two different paradigms for accessing its contents.

5.1 Appending Data to a Vector

- Use the vector constructor (c) to construct a vector with the additional data items.

```
> v <- c(v,newItems)

> v[length(v)+1] <- newItem


> v <- c(1,2,3)
> v <- c(v,4)          # Append a single value to v
> v
[1] 1 2 3 4
> w <- c(5,6,7,8)
> v <- c(v,w)          # Append an entire vector to v
> v
[1] 1 2 3 4 5 6 7 8
```

5.1 Appending Data to a Vector

- You can assign to any element and R will expand the vector to accommodate your request.

```
> v <- c(1,2,3)      # Create a vector of three elements
> v[10] <- 10        # Assign to the 10th element
> v                  # R extends the vector automatically
 [1]  1  2  3 NA NA NA NA NA NA 10
```

5.2 Inserting Data into a Vector

- Despite its name, the append function inserts data into a vector by using the after parameter, which gives the insertion point for the new item or items.

```
> append(vec, newvalues, after=n)
```

- The new items will be inserted at the position given by after.
- This example inserts 99 into the middle of a sequence.

```
> append(1:10, 99, after=5)
 [1]  1  2  3  4  5 99  6  7  8  9 10
```

- The special value of after=0 means insert the new items at the head of the vector.

```
> append(1:10, 99, after=0)
 [1] 99  1  2  3  4  5  6  7  8  9 10
```

5.3 Understanding the Recycling Rule

- Its useful to visualize the Recycling Rule. Here is a diagram of two vectors, 1:6 and 1:3

```
1:6 1:3
1   1
2   2
3   3
4
5
6
```

5.3 Understanding the Recycling Rule

- R recycles the elements of 1:3, pairing the two vectors like this and producing a six-element vector:

```
1:6 1:3 (1:6) + (1:3)
1   1   2
2   2   4
3   3   6
4   1   5
5   2   7
6   3   9
```

- Here is what you see at the command line.

```
> (1:6) + (1:3)
[1] 2 4 6 5 7 9
```

5.3 Understanding the Recycling Rule

- The cbind function can create column vectors, such as the following column vectors of 1:6 and 1:3.

```
> cbind(1:6)
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
[6,]    6
> cbind(1:3)
     [,1]
[1,]    1
[2,]    2
[3,]    3
```

5.3 Understanding the Recycling Rule

- The 1:3 vector is too short, so cbind invokes the Recycling Rule and recycles the elements of 1:3.

```
> cbind(1:6, 1:3)
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
[4,]    4    1
[5,]    5    2
[6,]    6    3
```

- If the longer vectors length is not a multiple of the shorter vectors length, R gives a warning.

```
> (1:6) + (1:5)          # Oops! 1:5 is one element too short
[1] 2 4 6 8 10 7
Warning message:
In (1:6) + (1:5) :
  longer object length is not a multiple of shorter object length
```

5.3 Understanding the Recycling Rule

- In this example, the 10 is recycled repeatedly until the vector addition is complete.

```
> (1:6) + 10
[1] 11 12 13 14 15 16
```

5.4 Creating a Factor (Categorical Variable)

- The factor function encodes your vector of discrete values into a factor.

```
> f <- factor(v)      # v is a vector of strings or integers

> f <- factor(v, levels)


> f <- factor(c("Win","Win","Lose","Tie","Win","Lose"))
> f
[1] Win Win Lose Tie Win Lose
Levels: Lose Tie Win
```

5.4 Creating a Factor (Categorical Variable)

- Suppose you have a string-valued variable wday that gives the day of the week on which your data was observed.

```
> f <- factor(wday)
> f
 [1] Wed Thu Mon Wed Thu Thu Thu Tue Thu Tue
Levels: Mon Thu Tue Wed

> f <- factor(wday, c("Mon","Tue","Wed","Thu","Fri"))
> f
 [1] Wed Thu Mon Wed Thu Thu Thu Tue Thu Tue
Levels: Mon Tue Wed Thu Fri
```

5.5 Combining Multiple Vectors into One Vector and a Factor

- Create a list that contains the vectors. Use the stack function to combine the list into a two-column data frame.

```
> comb <- stack(list(v1=v1, v2=v2, v3=v3))    # Combine 3 vectors
```

5.5 Combining Multiple Vectors into One Vector and a Factor

- You can combine the groups using the stack function.

```
> comb <- stack(list(fresh=freshmen, soph=sophomores, jrs=juniors))
> print(comb)
values ind
1 0.60 fresh
2 0.35 fresh
3 0.44 fresh
4 0.62 fresh
5 0.60 fresh
6 0.70 soph
7 0.61 soph
8 0.63 soph
9 0.87 soph
10 0.85 soph
11 0.70 soph
12 0.64 soph
```

5.5 Combining Multiple Vectors into One Vector and a Factor

- Now you can perform the ANOVA analysis on the two columns.

```
> aov(values ~ ind, data=comb)
```

5.6 Creating a List

- To create a list from individual data items, use the list function

```
> lst <- list(x, y, z)
```

- Lists can be quite simple, such as this list of three numbers

```
> lst <- list(0.5, 0.841, 0.977)
> lst
[[1]]
[1] 0.5

[[2]]
[1] 0.841

[[3]]
[1] 0.977
```

5.6 Creating a List

- Here is an extreme example of a mongrel created from a scalar, a character string, a vector, and a function.

```
> lst <- list(3.14, "Moe", c(1,1,2,3), mean)
> lst
[[1]]
[1] 3.14

[[2]]
[1] "Moe"

[[3]]
[1] 1 1 2 3

[[4]]
function (x, ...)
UseMethod("mean")
<environment: namespace:base>
```

5.6 Creating a List

- You can also build a list by creating an empty list and populating it. Here is our mongrel example built in that way.

```
> lst <- list()
> lst[[1]] <- 3.14
> lst[[2]] <- "Moe"
> lst[[3]] <- c(1,1,2,3)
> lst[[4]] <- mean
```

- List elements can be named. The list function lets you supply a name for every element.

```
> lst <- list(mid=0.5, right=0.841, far.right=0.977)
> lst
$mid
[1] 0.5

$right
[1] 0.841

$far.right
[1] 0.977
```

5.7 Selecting List Elements by Position

- Suppose we have a list of four integers, called years.

```
> years <- list(1960, 1964, 1976, 1994)
> years
[[1]]
[1] 1960

[[2]]
[1] 1964

[[3]]
[1] 1976

[[4]]
[1] 1994
```

5.7 Selecting List Elements by Position

- We can access single elements using the double-square-bracket syntax

```
> years[[1]]
[1] 1960
```

- We can extract sublists using the single-square-bracket syntax.

```
> years[c(1,2)]
[[1]]
[1] 1960

[[2]]
[1] 1964
```

5.7 Selecting List Elements by Position

- there is an important difference between lst[[n]] and lst[n].

```
> class(years[[1]])
[1] "numeric"

> class(years[1])
[1] "list"
```

5.7 Selecting List Elements by Position

- Recall that cat can print atomic values or vectors but complains about printing structured objects.

```
> cat(years[[1]], "\n")
1960
> cat(years[1], "\n")
Error in cat(list(...), file, sep, fill, labels, append) :
 argument 1 (type 'list') cannot be handled by 'cat'
```

5.8 Selecting List Elements by Name

- Each element of a list can have a name.
- This assignment creates a list of four named integers.

```
> years <- list(Kennedy=1960, Johnson=1964, Carter=1976, Clinton=1994)

> years[["Kennedy"]]
[1] 1960
> years$Kennedy
[1] 1960
```

5.8 Selecting List Elements by Name

- The following two expressions return sublists extracted from years.

```
> years[c("Kennedy","Johnson")]
$Kennedy
[1] 1960

$Johnson
[1] 1964

> years["Carter"]
$Carter
[1] 1976
```

5.9 Building a Name/Value Association List

- The list function lets you give names to elements, creating an association between each name and its value.

```
> lst <- list(mid=0.5, right=0.841, far.right=0.977)
```

- If you have parallel vectors of names and values, you can create an empty list and then populate the list by using a vectorized assignment statement.

```
> lst <- list()
> lst[names] <- values
```

5.9 Building a Name/Value Association List

- You can assign element names when you build the list. The list function allows arguments of the form name=value.

```
> lst <- list(
+             far.left=0.023,
+             left=0.159,
+             mid=0.500,
+             right=0.841,
+             far.right=0.977)
> lst
$far.left
[1] 0.023
$left
[1] 0.159
$mid
[1] 0.5
$right
[1] 0.841
$far.right
[1] 0.977
```

5.9 Building a Name/Value Association List

- One way to name the elements is to create an empty list and then populate it via assignment statements.

```
> lst <- list()
> lst$far.left <- 0.023
> lst$left <- 0.159
> lst$mid <- 0.500
> lst$right <- 0.841
> lst$far.right <- 0.977
```

- Sometimes you have a vector of names and a vector of corresponding values.

```
> values <- pnorm(-2:2)
> names <- c("far.left", "left", "mid", "right", "far.right")
```

5.9 Building a Name/Value Association List

- You can associate the names and the values by creating an empty list and then populating it with a vectorized assignment statement.

```
> lst <- list()
> lst[names] <- values
> lst
$far.left
[1] 0.02275013
$left
[1] 0.1586553
$mid
[1] 0.5
$right
[1] 0.8413447
$far.right
[1] 0.9772499
```

5.9 Building a Name/Value Association List

- Once the association is made, the list can translate names into values through a simple list lookup.

```
> cat("The left limit is", lst[["left"]], "\n")
The left limit is 0.1586553
> cat("The right limit is", lst[["right"]], "\n")
The right limit is 0.8413447

> for (nm in names(lst)) cat("The", nm, "limit is", lst[[nm]], "\n")
The far.left limit is 0.02275013
The left limit is 0.1586553
The mid limit is 0.5
The right limit is 0.8413447
The far.right limit is 0.9772499
```

5.10 Removing an Element from a List

- To remove a list element, select it by position or by name, and then assign NULL to the selected element.

```
> years
$Kennedy
[1] 1960
$Johnson
[1] 1964
$Carter
[1] 1976
$Clinton
[1] 1994
> years[["Johnson"]] <- NULL          # Remove the element labeled "Johnson"
> years
$Kennedy
[1] 1960
$Carter
[1] 1976
$Clinton
[1] 1994
```

5.10 Removing an Element from a List

- You can remove multiple elements this way, too.

```
> years[c("Carter","Clinton")] <- NULL # Remove two elements
> years
$Kennedy
[1] 1960
```

5.11 Flatten a List into a Vector

- Basic statistical functions work on vectors but not on lists, for example. If iq.scores is a list of numbers, then we cannot directly compute their mean.

```
> mean(iq.scores)
[1] NA
Warning message:
In mean.default(iq.scores) :
  argument is not numeric or logical: returning NA
```

- Instead, we must flatten the list into a vector using unlist and then compute the mean of the result.

```
> mean(unlist(iq.scores))
[1] 106.4452
```

5.11 Flatten a List into a Vector

- Here is another example. We can cat scalars and vectors, but we cannot cat a list

```
> cat(iq.scores, "\n")
Error in cat(list(...), file, sep, fill, labels, append) :
  argument 1 (type 'list') cannot be handled by 'cat'
```

- One solution is to flatten the list into a vector before printing

```
> cat("IQ Scores:", unlist(iq.scores), "\n")
IQ Scores: 89.73383 116.5565 113.0454
```

5.12 Removing NULL Elements from a List

- Suppose lst is a list some of whose elements are NULL. This expression will remove the NULL elements.

```
> lst[sapply(lst, is.null)] <- NULL
```

5.12 Removing NULL Elements from a List

- The curious reader may be wondering how a list can contain NULL elements, given that we remove elements by setting them to NULL (Recipe 5.10).

```
> lst <- list("Moe", NULL, "Curly") # Create list with NULL element
> lst
[[1]]
[1] "Moe"
[[2]]
NULL
[[3]]
[1] "Curly"
> lst[sapply(lst, is.null)] <- NULL # Remove NULL element from list
> lst
[[1]]
[1] "Moe"
[[2]]
[1] "Curly"
```

5.13 Removing List Elements Using a Condition

- This assignment, for example, removes all negative value from lst.

```
> lst[lst < 0] <- NULL
```

```
> lst[lst == 0] <- NULL
```

- This expression will remove NA values from the list.

```
> lst[is.na(lst)] <- NULL
```

5.13 Removing List Elements Using a Condition

```
> lst[abs(lst) < 1] <- NULL
Error in abs(lst) : non-numeric argument to function

> lst[abs(unlist(lst)) < 1] <- NULL

> lst[lapply(lst,abs) < 1] <- NULL

> mods[sapply(mods, function(m) summary(m)$r.squared < 0.3)] <- NULL
```

5.14 Initializing a Matrix

- This example shapes a vector into a 2 3 matrix (i.e., two rows and three columns).

```
> matrix(vec, 2, 3)
```

- Suppose we want to create and initialize a 2 3 matrix.
- We can capture the initial data inside a vector and then shape it using the matrix function.

```
> theData <- c(1.1, 1.2, 2.1, 2.2, 3.1, 3.2)
> mat <- matrix(theData, 2, 3)
> mat
     [,1] [,2] [,3]
[1,] 1.1  2.1  3.1
[2,] 1.2  2.2  3.2
```

5.14 Initializing a Matrix

- Its common to initialize an entire matrix to one value such as zero or NA.
- If the first argument of matrix is a single value, then R will apply the Recycling Rule and automatically replicate the value to fill the entire matrix.

```
> matrix(0, 2, 3)          # Create an all-zeros matrix
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
> matrix(NA, 2, 3)         # Create a matrix populated with NA
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
```

5.14 Initializing a Matrix

- You can create a matrix with a one-liner, of course, but it becomes difficult to read.

```
> mat <- matrix(c(1.1, 1.2, 1.3, 2.1, 2.2, 2.3), 2, 3)
> mat
     [,1] [,2] [,3]
[1,]  1.1  1.3  2.2
[2,]  1.2  2.1  2.3
```

5.14 Initializing a Matrix

- A common idiom in R is typing the data itself in a rectangular shape that reveals the matrix structure.

```
> theData <- c(1.1, 1.2, 1.3,
+              2.1, 2.2, 2.3)
> theData
[1] 1.1 1.2 1.3 2.1 2.2 2.3
> mat <- matrix(theData, 2, 3, byrow=TRUE)
> mat
     [,1] [,2] [,3]
[1,]  1.1  1.2  1.3
[2,]  2.1  2.2  2.3
```

5.14 Initializing a Matrix

- Setting byrow=TRUE tells matrix that the data is row-by-row and not column-by-column (which is the default). In condensed form, that becomes.

```
> mat <- matrix(c(1.1, 1.2, 1.3,
+                 2.1, 2.2, 2.3),
+               2, 3, byrow=TRUE)
> mat
     [,1] [,2] [,3]
[1,]  1.1  1.2  1.3
[2,]  2.1  2.2  2.3
```

- The following example creates a vanilla vector and then shapes it into a 2 3 matrix.

```
> v <- c(1.1, 1.2, 1.3, 2.1, 2.2, 2.3)
> dim(v) <- c(2,3)
> v
     [,1] [,2] [,3]
[1,]  1.1  1.3  2.2
[2,]  1.2  2.1  2.3
```

5.15 Performing Matrix Operations

- t(A): Matrix transposition of A
- solve(A): Matrix inverse of A
- A %% B: Matrix multiplication of A and B
- diag(n): An n-by-n diagonal (identity) matrix

5.16 Giving Descriptive Names to the Rows and Columns of a Matrix

- Assign a vector of character strings to the appropriate attribute.

```
> rownames(mat) <- c("rowname1", "rowname2", ..., "rownamem")
> colnames(mat) <- c("colname1", "colname2", ..., "colnamen")
```

- Consider this matrix of correlations between the prices of IBM, Microsoft, and Google stock.

```
> print(tech.corr)
      [,1]  [,2]  [,3]
[1,] 1.000 0.556 0.390
[2,] 0.556 1.000 0.444
[3,] 0.390 0.444 1.000
```

5.16 Giving Descriptive Names to the Rows and Columns of a Matrix

```
> colnames(tech.corr) <- c("IBM","MSFT","GOOG")
> rownames(tech.corr) <- c("IBM","MSFT","GOOG")
> print(tech.corr)
       IBM  MSFT  GOOG
IBM  1.000 0.556 0.390
MSFT 0.556 1.000 0.444
GOOG 0.390 0.444 1.000
```

- Another advantage of naming rows and columns is that you can refer to matrix elements by those names.

```
> tech.corr["IBM","GOOG"]        # What is the correlation between IBM and GOOG?
[1] 0.39
```

5.17 Selecting One Row or Column from a Matrix

- The solution depends on what you want. If you want the result to be a simple vector, just use normal indexing.

```
> vec <- mat[1,]          # First row
> vec <- mat[,3]          # Third column
```

- If you want the result to be a one-row matrix or a one-column matrix, then include the drop=FALSE argument.

```
> row <- mat[1,,drop=FALSE]          # First row in a one-row matrix
> col <- mat[,3,drop=FALSE]          # Third column in a one-column matrix
```

5.17 Selecting One Row or Column from a Matrix

- Normally, when you select one row or column from a matrix, R strips off the dimensions.

```
> mat[1,]
[1] 1 4 7 10
> mat[,3]
[1] 7 8 9
```

- When you include the drop=FALSE argument, however, R retains the dimensions. In that case, selecting a row returns a row vector (a 1 n matrix).

```
> mat[1,,drop=FALSE]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
```

- Likewise, selecting a column with drop=FALSE returns a column vector (an n 1 matrix).

```
> mat[,3,drop=FALSE]
     [,1]
[1,]    7
[2,]    8
[3,]    9
```

5.18 Initializing a Data Frame from Column Data

- If your data is captured in several vectors and/or factors, use the data.frame function to assemble them into a data frame.

```
> dfrm <- data.frame(v1, v2, v3, f1, f2)
```

- If your data is captured in a list that contains vectors and/or factors, use instead as.data.frame.

```
> dfrm <- as.data.frame(list.of.vectors)
```

- The data.frame function can construct a data frame from vectors, where each vector is one observed variable.

5.18 Initializing a Data Frame from Column Data

- Suppose you have two numeric predictor variables, one categorical predictor variable, and one response variable.
- The data.frame function can create a data frame from your vectors.

```
> dfrm <- data.frame(pred1, pred2, pred3, resp)
> dfrm
        pred1       pred2 pred3     resp
1  -2.7528917 -1.40784130    AM 12.57715
2  -0.3626909  0.31286963    AM 21.02418
3  -1.0416039 -0.69685664    PM 18.94694
4   1.2666820 -1.27511434    PM 18.98153
5   0.7806372 -0.27292745    AM 19.59455
6  -1.0832624  0.73383339    AM 20.71605
7  -2.0883305  0.96816822    PM 22.70062
8  -0.7063653 -0.84476203    PM 18.40691
9  -0.8394022  0.31530793    PM 21.00930
10 -0.4966884 -0.08030948    AM 19.31253
```

5.18 Initializing a Data Frame from Column Data

- Notice that data.frame takes the column names from your program variables.
- You can override that default by supplying explicit column names.

```
> dfrm <- data.frame(p1=pred1, p2=pred2, p3=pred3, r=resp)
> dfrm
          p1          p2 p3       r
1 -2.7528917 -1.40784130 AM 12.57715
2 -0.3626909  0.31286963 AM 21.02418
3 -1.0416039 -0.69685664 PM 18.94694
.
. (etc.)
.
```

5.18 Initializing a Data Frame from Column Data

- Alternatively, your data may be organized into vectors but those vectors are held in a list, not individual program variables, like this:

```
> lst <- list(p1=pred1, p2=pred2, p3=pred3, r=resp)
```

- No problem. Use the as.data.frame function to create a data frame from the list of vectors.

```
> as.data.frame(lst)
          p1          p2 p3        r
1 -2.7528917 -1.40784130 AM 12.57715
2 -0.3626909  0.31286963 AM 21.02418
3 -1.0416039 -0.69685664 PM 18.94694
.
. (etc.)
.
```

5.19 Initializing a Data Frame from Row Data

- Store each row in a one-row data frame. Store the one-row data frames in a list. Use rbind and do.call to bind the rows into one, large data frame.

```
> dfrm <- do.call(rbind, obs)
```

- Here, obs is a list of one-row data frames.
- Those data frames are stored in a list called obs. The first element of obs might look like this:

```
> obs[[1]]
   pred1 pred2 pred3   resp
1 -1.197  0.36    AM 18.701
```

5.19 Initializing a Data Frame from Row Data

- We need to bind together those rows into a data frame.
- If we rbind the first two observations, for example, we get a two-row data frame:

```
> rbind(obs[[1]], obs[[2]])
   pred1 pred2 pred3   resp
1 -1.197  0.36    AM 18.701
2 -0.952  1.23    PM 25.709
```

5.19 Initializing a Data Frame from Row Data

- The do.call function will expand obs into one, long argument list and call rbind with that long argument list.

```
> do.call(rbind,obs)
   pred1  pred2 pred3   resp
1 -1.197  0.360    AM 18.701
2 -0.952  1.230    PM 25.709
3  0.279  0.423    PM 21.572
4 -1.445 -1.846    AM 14.392
5  0.822 -0.246    AM 19.841
6  1.247  1.254    PM 25.637
7 -0.394  1.563    AM 24.585
8 -1.248 -1.264    PM 16.770
```

- We first transform the rows into data frames using the Map function and then apply this recipe.

```
> dfrm <- do.call(rbind,Map(as.data.frame,obs))
```

5.20 Appending Rows to a Data Frame

- Suppose we want to append a new row to our data frame of Chicago-area cities.
- First, we create a one-row data frame with the new data.

```
> newRow <- data.frame(city="West Dundee", county="Kane", state="IL", pop=5428)
```

- Next, we use the rbind function to append that one-row data frame to our existing data frame.

```
> suburbs <- rbind(suburbs, newRow)
> suburbs
              city   county state     pop
1          Chicago     Cook    IL 2853114
2          Kenosha  Kenosha    WI   90352
3           Aurora     Kane    IL  171782
4            Elgin     Kane    IL   94487
5             Gary Lake(IN)    IN  102746
6           Joliet  Kendall    IL  106221
7       Naperville   DuPage    IL  147779
```

5.20 Appending Rows to a Data Frame

- We can combine these two steps into one, of course.

```
> suburbs <- rbind(suburbs,
+ data.frame(city="West Dundee", county="Kane", state="IL", pop=5428))
```

- We can even extend this technique to multiple new rows because rbind allows multiple arguments.

```
> suburbs <- rbind(suburbs,
+ data.frame(city="West Dundee", county="Kane", state="IL", pop=5428),
+ data.frame(city="East Dundee", county="Kane", state="IL", pop=2955))
```

5.21 Preallocating a Data Frame

- Create a data frame from generic vectors and factors using the functions numeric(n), character(n), and factor(n).

```
> dfrm <- data.frame(colname1=numeric(n), colname2=character(n), ... etc. ... )
```

- Suppose you want to create a data frame with 1,000,000 rows and three columns: two numeric and one character.
- Use the numeric and character functions to preallocate the columns; then join them together using data.frame.

```
> N <- 1000000
> dfrm <- data.frame(dosage=numeric(N), lab=character(N), response=numeric(N))
> dfrm
   dosage lab response
1       0             0
2       0             0
3       0             0
4       0             0
.
. (etc.)
.
```

5.21 Preallocating a Data Frame

- You cant simply call factor(n).
- Continuing our example, suppose you want the lab column to be a factor, not a character string, and that the possible levels are NJ, IL, and CA. Include the levels in the column specification, like this:

```
> N <- 1000000
> dfrm <- data.frame(dosage=numeric(N),
+                    lab=factor(N, levels=c("NJ", "IL", "CA")),
+                    response=numeric(N) )
> dfrm
   dosage  lab response
1       0 <NA>        0
2       0 <NA>        0
3       0 <NA>        0
4       0 <NA>        0
.
. (etc.)
.
```

5.22 Selecting Data Frame Columns by Position

- Lets play with the population data for the 16 largest cities in the Chicago metropolitan area.

```
> suburbs
        city    county state     pop
1    Chicago      Cook    IL 2853114
2    Kenosha  Kenosha     WI   90352
3     Aurora      Kane    IL  171782
4      Elgin      Kane    IL   94487
5       Gary Lake(IN)     IN  102746
6     Joliet  Kendall     IL  106221
7 Naperville   DuPage     IL  147779
.
. (etc.)
.
```

5.22 Selecting Data Frame Columns by Position

- Use simple list notation to select exactly one column, such as the first column.

```
> suburbs[[1]]
[1] "Chicago"     "Kenosha"    "Aurora"     "Elgin"
[5] "Gary"        "Joliet"     "Naperville" "Arlington Heights"
[9] "Bolingbrook" "Cicero"     "Evanston"   "Hammond"
[13] "Palatine"   "Schaumburg" "Skokie"     "Waukegan"
```

- The first column of suburbs is a vector, so thats what suburbs[[1]] returns: a vector.

5.22 Selecting Data Frame Columns by Position

- The result differs when you use the single-bracket notation, as in suburbs[1] or suburbs[c(1,3)].
- This example returns the first column wrapped in a data frame:

```
> suburbs[1]
city
1 Chicago
2 Kenosha
3 Aurora
4 Elgin
5 Gary
6 Joliet
7 Naperville
.
. (etc.)
.
```

5.22 Selecting Data Frame Columns by Position

- The next example returns the first and third columns wrapped in a data frame:

```
> suburbs[c(1,3)]
city pop
1 Chicago 2853114
2 Kenosha 90352
3 Aurora 171782
4 Elgin 94487
5 Gary 102746
6 Joliet 106221
7 Naperville 147779
.
. (etc.)
.
```

5.22 Selecting Data Frame Columns by Position

- A major source of confusion is that suburbs[[1]] and suburbs[1] look similar but produce very different results:
  - suburbs[[1]]: This returns one column.
  - suburbs[1]: This returns a data frame, and the data frame contains exactly one column. This is a special case of dfrm[c($n_1, n_2, ..., n_k$)]. We dont need the c(...) construct because there is only one n.
- n the simple case of one index you get a column, like this:

```
> suburbs[,1]
 [1] "Chicago"     "Kenosha"    "Aurora"     "Elgin"
 [5] "Gary"        "Joliet"     "Naperville" "Arlington Heights"
 [9] "Bolingbrook" "Cicero"     "Evanston"   "Hammond"
[13] "Palatine"    "Schaumburg" "Skokie"     "Waukegan"
```

5.22 Selecting Data Frame Columns by Position

- But using the same matrix-style syntax with multiple indexes returns a data frame.

```
> suburbs[,c(1,4)]
        city     pop
1    Chicago 2853114
2    Kenosha   90352
3     Aurora  171782
4      Elgin   94487
5       Gary  102746
6     Joliet  106221
7 Naperville  147779
.
. (etc.)
.
```

5.22 Selecting Data Frame Columns by Position

- This creates a problem. Suppose you see this expression in some old R script

```
dfrm[,vec]
```

- To avoid this problem, you can include drop=FALSE in the subscripts; this forces R to return a data frame.

```
dfrm[,vec,drop=FALSE]
```

5.23 Selecting Data Frame Columns by Name

- To select a single column, use one of these list expressions:

  dfrm[["name"]]: Returns one column, the column called name.

  dfrm$name: Same as previous, just different syntax.

- To select one or more columns and package them in a data frame, use these list expressions:

  dfrm["name"] Selects one column and packages it inside a data frame object.

  dfrm[c("$name_1$", "$name_2$", ..., "$name_k$")] Selects several columns and packages them in a data frame.

- You can use matrix-style subscripting to select one or more columns:

  dfrm[, "name"]: Returns the named column.

  dfrm[, c("$name_1$", "$name_2$", ..., "$name_k$")]: Selects several columns and packages in a data frame.

5.24 Selecting Rows and Columns More Easily

- Use the subset function. The select argument is a column name, or a vector of column names, to be selected.

```
> subset(dfrm, select=colname)
> subset(dfrm, select=c(colname1, ..., colnameN))
```

- Note that you do not quote the column names.
- In this example, response is a column in the data frame, and we are selecting rows with a positive response.

```
> subset(dfrm, subset=(response > 0))
```

- subset is most useful when you combine the select and subset arguments.

```
> subset(dfrm, select=c(predictor,response), subset=(response > 0))
```

5.24 Selecting Rows and Columns More Easily

- *Select the model name for cars that can exceed 30 miles per gallon (MPG) in the city.*

```
> subset(Cars93, select=Model, subset=(MPG.city > 30))
     Model
31 Festiva
39   Metro
42   Civic
.
. (etc.)
.
```

5.24 Selecting Rows and Columns More Easily

- *Select the model name and price range for four-cylinder cars made in the United States.*

```
> subset(Cars93, select=c(Model,Min.Price,Max.Price),
+        subset=(Cylinders == 4 & Origin == "USA"))
      Model Min.Price Max.Price
6   Century      14.2      17.3
12 Cavalier       8.5      18.3
13  Corsica      11.4      11.4
.
. (etc.)
.
```

5.24 Selecting Rows and Columns More Easily

- *Select the manufacturers name and the model name for all cars whose highway MPG value is above the median.*

```
> subset(Cars93, select=c(Manufacturer,Model),
+        subset=c(MPG.highway > median(MPG.highway)))
   Manufacturer   Model
1         Acura Integra
5           BMW    535i
6         Buick Century
.
. (etc.)
.
```

5.25 Changing the Names of Data Frame Columns

- Data frames have a colnames attribute that is a vector of column names. You can update individual names or the entire vector.

```
> colnames(dfrm) <- newnames      # newnames is a vector of character strings
```

- The columns of data frames must have names.

```
> mat
       [,1]   [,2]   [,3]
[1,] -0.818 -0.667 -0.494
[2,] -0.819 -0.946 -0.205
[3,]  0.385  1.531 -0.611
[4,] -2.155 -0.535 -0.316
> as.data.frame(mat)
      V1     V2     V3
1 -0.818 -0.667 -0.494
2 -0.819 -0.946 -0.205
3  0.385  1.531 -0.611
4 -2.155 -0.535 -0.316
```

5.25 Changing the Names of Data Frame Columns

- If the matrix had column names defined, R would have used those names instead of synthesizing new ones.
- However, converting a list into a data frame produces some strange synthetic names.

```
> lst
[[1]]
[1] -0.284 -1.114 -1.097 -0.873
[[2]]
[1] -1.673 0.929 0.306 0.778
[[3]]
[1] 0.323 0.368 0.067 -0.080
```

5.25 Changing the Names of Data Frame Columns

```
> as.data.frame(lst)
  c..0.284...1.114...1.097...0.873. c..1.673..0.929..0.306..0.778.
1                            -0.284                          -1.673
2                            -1.114                           0.929
3                            -1.097                           0.306
4                            -0.873                           0.778
  c.0.323..0.368..0.067...0.08.
1                         0.323
2                         0.368
3                         0.067
4                        -0.080
```

5.25 Changing the Names of Data Frame Columns

- Fortunately, you can overwrite the synthetic names with names of your own by setting the colnames attribute.

```
> dfrm <- as.data.frame(lst)
> colnames(dfrm) <- c("before","treatment","after")
> dfrm
  before treatment  after
1 -0.284    -1.673  0.323
2 -1.114     0.929  0.368
3 -1.097     0.306  0.067
4 -0.873     0.778 -0.080
```

5.26 Editing a Data Frame

- R includes a data editor that displays your data frame in a spreadsheet-like window. Invoke the editor using the edit function:

```
> temp <- edit(dfrm)
> dfrm <- temp # Overwrite only if you're happy with the changes!
```

- If you are feeling brave, the fix function invokes the editor and overwrites your variable with the result. There is no undo, however:

```
> fix(dfrm)
```

5.27 Removing NAs from a Data Frame

- Here we can see cumsum fail because the input contains NA values:

```
> dfrm
x y
1 -0.9714511 -0.4578746
2 NA 3.1663282
3 0.3367627 NA
4 1.7520504 0.7406335
5 0.4918786 1.4543427
> cumsum(dfrm)
x y
1 -0.971451 -0.4578746
2 NA 2.7084536
3 NA NA
4 NA NA
5 NA NA
```

5.27 Removing NAs from a Data Frame

- If we remove the NA values, cumsum can complete its summations:

```
> cumsum(na.omit(dfrm))
x y
1 -0.9714511 -0.4578746
4 0.7805993 0.2827589
5 1.2724779 1.7371016
```

- This recipe works for vectors and matrices, too, but not for lists.

5.28 Excluding Columns by Name

- Use the subset function with a negated argument for the select parameter:

```
> subset(dfrm, select = -badboy) # All columns except badboy
```

- I often encounter this problem when calculating the correlation matrix of a data frame and I want to exclude nondata columns such as labels:

```
> cor(patient.data)
            patient.id         pre     dosage        post
patient.id  1.00000000  0.02286906  0.3643084 -0.13798149
pre         0.02286906  1.00000000  0.2270821 -0.03269263
dosage      0.36430837  0.22708208  1.0000000 -0.42006280
post       -0.13798149 -0.03269263 -0.4200628  1.00000000
```

5.28 Excluding Columns by Name

- We can exclude the patient ID column to clean up the output:

```
> cor(subset(patient.data, select = -patient.id))
               pre     dosage        post
pre     1.00000000  0.2270821 -0.03269264
dosage  0.22708207  1.0000000 -0.42006280
post   -0.03269264 -0.4200628  1.00000000
```

- We can exclude multiple columns by giving a vector of negated names:

```
> cor(subset(patient.data, select = c(-patient.id,-dosage)))
              pre        post
pre   1.00000000 -0.03269264
post -0.03269264  1.00000000
```

5.29 Combining Two Data Frames

- The cbind function will combine data frames side by side, as shown here when combining stooges and birth:

```
> stooges
   name n.marry n.child
1   Moe       1       2
2 Larry       1       2
3 Curly       4       2
> birth
  birth.year  birth.place
1       1887  Bensonhurst
2       1902 Philadelphia
3       1903     Brooklyn
> cbind(stooges,birth)
   name n.marry n.child birth.year  birth.place
1   Moe       1       2       1887  Bensonhurst
2 Larry       1       2       1902 Philadelphia
3 Curly       4       2       1903     Brooklyn
```

5.29 Combining Two Data Frames

- The rbind function will stack the rows of two data frames, as shown here when combining stooges and guys:

```
> stooges
   name n.marry n.child
1   Moe       1       2
2 Larry       1       2
3 Curly       4       2
> guys
   name n.marry n.child
1   Tom       4       2
2  Dick       1       4
3 Harry       1       1
```

5.29 Combining Two Data Frames

- ```
  > rbind(stooges,guys)
     name n.marry n.child
  1   Moe       1       2
  2 Larry       1       2
  3 Curly       4       2
  4   Tom       4       2
  5  Dick       1       4
  6 Harry       1       1
  ```
  ```

5.30 Merging Data Frames by Common Column

- Use the merge function to join the data frames into one new data frame based on the common column:

```
> m <- merge(df1, df2, by="name")
```

- Suppose you have two data frames, born and died, that each contain a column called

```
name:
    > born
       name year.born   place.born
    1   Moe      1887  Bensonhurst
    2 Larry      1902 Philadelphia
    3 Curly      1903     Brooklyn
    4 Harry      1964       Moscow
    > died
       name year.died
    1 Curly      1952
    2   Moe      1975
    3 Larry      1975
```

5.30 Merging Data Frames by Common Column

- We can merge them into one data frame by using name to combine matched rows:

```
> merge(born, died, by="name")
   name year.born   place.born year.died
1 Curly      1903     Brooklyn      1952
2 Larry      1902 Philadelphia      1975
3   Moe      1887  Bensonhurst      1975
```

- Notice that merge does not require the rows to be sorted or even to occur in the same order.
- It found the matching rows for Curly even though they occur in different positions.

5.31 Accessing Data Frame Contents More Easily

- For quick, one-off expressions, use the with function to expose the column names:

```
> with(dataframe, expr)
```

- You can then refer to the data frame columns by name without mentioning the data frame:

```
> attach(dataframe)
```

- Use the detach function to remove the data frame from your search list.

5.31 Accessing Data Frame Contents More Easily

- For a data frame called suburbs that contains a column called pop, here is the nave way to calculate the z-scores of pop:

```
> z <- (suburbs$pop - mean(suburbs$pop)) / sd(suburbs$pop)
```

- you can refer to the data frame columns by their names:

```
> z <- with(suburbs, (pop - mean(pop)) / sd(pop))
```

5.31 Accessing Data Frame Contents More Easily

- If you will be working repeatedly with columns in your data frame, attach the data frame to your search list and the columns will become available as variables:

```
> attach(suburbs)
```

- After the attach, the second item in the search list is the suburbs data frame:

```
> search()
 [1] ".GlobalEnv"        "suburbs"           "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

- Now we can refer to the columns of the data frame as if they were variables:

```
> z <- (pop - mean(pop)) / sd(pop)
```

5.31 Accessing Data Frame Contents More Easily

- When you are done, use a detach (with no arguments) to remove the second location in the search list:

```
> detach()
> search()
[1] ".GlobalEnv"        "package:stats" "package:graphics"
[4] "package:grDevices" "package:utils" "package:datasets"
[7] "package:methods"   "Autoloads"     "package:base"
```

5.31 Accessing Data Frame Contents More Easily

- In this session fragment, notice how changing the data frame does not change our view of the attached data:

```
> attach(suburbs)
> pop
 [1] 2853114 90352 171782 94487 102746 106221 147779 76031 70834
[10]   72616 74239  83048 67232  75386  63348  91452
> suburbs$pop <- 0      # Overwrite data frame contents
> pop                   # Hey! It seems nothing changed
 [1] 2853114 90352 171782 94487 102746 106221 147779 76031 70834
[10]   72616 74239  83048 67232  75386  63348  91452
> suburbs$pop           # Contents of data frame did indeed change
[1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

5.31 Accessing Data Frame Contents More Easily

- In the following fragment, you might think we are scaling pop by 1,000 but we are actually creating a new local variable:

```
> attach(suburbs)
> pop
 [1] 2853114 90352 171782 94487 102746 106221 147779 76031 70834
[10]   72616 74239  83048 67232  75386  63348  91452
> pop <- pop / 1000   # Achtung! This is creating a local variable
  called "pop"
> ls()                # We can see the new variable in our workspace
[1] "pop" "suburbs"
> suburbs$pop         # Original data is unchanged
 [1] 2853114 90352 171782 94487 102746 106221 147779 76031 70834
[10]   72616 74239  83048 67232  75386  63348  91452
```

5.32 Converting One Atomic Value into Another

- Converting one atomic type into another is usually pretty simple. If the conversion works, you get what you would expect. If it does not work, you get NA:

```
> as.numeric(" 3.14 ")
[1] 3.14
> as.integer(3.14)
[1] 3
> as.numeric("foo")
[1] NA
Warning message:
NAs introduced by coercion
> as.character(101)
[1] "101"
```

5.32 Converting One Atomic Value into Another

- ```
  > as.numeric(c("1","2.718","7.389","20.086"))
  [1] 1.000 2.718 7.389 20.086
  > as.numeric(c("1","2.718","7.389","20.086", "etc."))
  [1] 1.000 2.718 7.389 20.086 NA
  Warning message:
  NAs introduced by coercion
  > as.character(101:105)
  [1] "101" "102" "103" "104" "105"
  ```
  ```