Chapter 5:
Data Manipulation

Vector Operations

- ✓ Subscripts of a vector
  - ➡ A part of a vector can be returned by using a subscript
    - Usage: Type *vector*[{subscript}] at the command line
    - Example: x[3] will return 5, where x = 3 2 5 9 1
  - ➡ Specifying a sequence in the subscript returns the part of the vector where the position of the returned elements is as specified in the sequence
    - Usage: Type *vector*[{sequence}] at the command line
    - Example: In the above example, x[2:4] will return 2 5 9
  - ➡ Specifying a vector (B) in the subscript returns the part of the vector (A) where the position of the returned elements is as specified in the vector (B)
    - Usage: Type *vector*[{vector}] at the command line
    - Example: In the above example, x[c(1,2,1)] will return 3 2 3
  - ➡ Specifying a logical expression in the subscript returns only those elements that satisfy the expression
    - Usage: Type *vector*[{logical expression}] at the command line
    - Example: In the above example, x[x > 3] will return 5 9
  - ➡ Specifying a vector of negative numbers will return all elements other than those specified in the vector
    - Usage: Type *vector*[-{vector}] at the command line
    - Example: In the above example, x[-c(1, 2, 3)] will return 9 1

Vector Operations

- ✓ Ordering
  - ➡ Function *order*() is used to return a vector which is the position of elements of the object if ordered
    - Usage: Type *order*({object}) at the command line
    - Example:
      - *order*(x) will return 3 1 2 5 4, where x = 2 2 1 4 3
      - x[*order*(x)] will return 1 2 2 3 4
  - ➡ Function *rank*() returns the rank of each element in the object. If a tie occurs - multiple elements of same value - a tie-breaker method can be specified
    - Usage: Type *rank*({object}, ties.method = ...) at the command line
    - Example:
      - *rank*(x, ties.method = "random") will return 2 3 1 5 4, where x = 2 2 1 4 3
  - ➡ Function *rev*() returns the vector elements in reverse order
    - Usage: Type *rev*({object) at the command line
    - Example:
      - *rev*(x) will return 3 4 1 2 2, where x = 2 2 1 4 3

Vector Operations

✓ Statistics

➡ Typically, statistics taken on a vector will yield a number

- Examples: If x = 1 2 4 2 1 4 7 2 1,
  - *length*(x) will yield 9
  - *sum*(x) will yield 24
  - *prod*(x) will yield 896
  - *min*(x) will yield 1
  - *max*(x) will yield 7
  - *mean*(x) will yield 2.7
  - *median*(x) will yield 2
  - *sd*(x) will yield 2
  - *cumsum*(x), the cumulative sum of x, will yield 1 3 7 9 10 14 21 23 24

```
> x <- c(1,2,4,2,1,4,7,2,1)
> length(x)
[1] 9
> sum(x)
[1] 24
> prod(x)
[1] 896
> min(x)
[1] 1
> max(x)
[1] 7
> mean(x)
[1] 2.666667
> median(x)
[1] 2
> sd(x)
[1] 2
> cumsum(x)
[1]  1  3  7  9 10 14 21 23 24
>
```

Vector Operations

- ✓ Applying functions
  - ➡ Function *lapply*() is used to apply a function to a vector (or list). Result is stored in a list
    - Usage: Type *lapply*({vector}, {function}) at the command line
    - Example: *lapply*(x, sqrt) will return 1 1.414 2 2.236 1.414 (as a list), where x = 1 2 4 5 2
  - ➡ Function *sapply*() is similar to *lapply*(x). However, result can be simplified into a vector
    - Usage: Type *sapply*({vector}, {function}, simplify = ...) at the command line
    - Example: *sapply*(x, sqrt) will return 1 1.414 2 2.236 1.414 (as a vector), in the above example
- ✓ Subdivision
  - ➡ Function *split*() is used to subdivide a vector into groups. Result is stored in a list.
    - Usage: Type *split*({vector}, {factor}, ...) at the command line
    - Examples:

```
> x <- c(1,1,2,4,2,1,1,2,2,2,4,4,1,1,2,4,2)
> y <- split(x,c(1,2,4))
Warning message:
In split.default(x, c(1, 2, 4)) :
  data length is not a multiple of split variable
> y
$`1`
[1] 1 4 1 2 1 4

$`2`
[1] 1 2 2 4 1 2

$`4`
[1] 2 1 2 4 2

>
```

```
> split(1:10, 1:2)
$`1`
[1] 1 3 5 7 9

$`2`
[1]  2  4  6  8 10

> split(1:12, 1:3)
$`1`
[1]  1  4  7 10

$`2`
[1]  2  5  8 11

$`3`
[1]  3  6  9 12
```

Vector Operations

✓ Sampling

➡ Function *sample*() is used to randomly sample a vector and return values of a certain sample size

- Usage: Type *sample*({vector}, {size}, replace = …, prob = …) at the command line (the *prob* parameter may be used to specify weights for the elements in the vector)
- Examples:

```
> x <- c(1,2,3,4,1,3,12,3,1,1)
> sample(x,2,replace = TRUE)
[1]  1 12
> sample(x,2,replace = TRUE)
[1] 4 2
> sample(x,5,replace = TRUE)
[1] 4 2 4 1 4
> sample(x,5,replace = FALSE)
[1]  3  1  1 12  2
>
```

Exercise

Array Operations

- ✓ Subscripts of an array
  - ➡ A part of an array can be returned by using a subscript
    - Usage: Type *array*[{subscript1}, {subscript2},…,{subscriptn}] at the command line to return a specific element. Subscript1, 2,...n refers to the dimensions of an array
    - Example: x[2,3] will return 6, where x = 1 2 3
      4 5 6
      7 8 9
    - If one of the subscripts is empty, the full range of that subscript is taken
    - Example: In the above example, x[2,] = 4 5 6
  - ➡ Methods used for vector subscripts also apply to arrays. These include
    - Specifying a sequence
    - Using a vector
    - Including a logical expression
    - Excluding elements with negative subscripts

Array Operations

- ✓ Outer product
  - ➡ Function *outer*() may be used to determine the outer product of two arrays.
    - Usage: Type *outer*[{array1}, {array 2},FUN = , …] at the command line to return a specific element.
    - The result is an array of dimensions c(dim(array1), dim(array2)).
      - For example, if X is a 2 x 3 array, and Y is a 3 x 4 array, XY = *outer*(X,Y) is a 2 x 3 x 3 x 4 array
    - Element XY[c(arrayindex.X, arrayindex.Y)] = FUN(X[arrayindex.X],Y[arrayindex.Y], …).
      - In the above example, if XY[2,1,2,3] = FUN(X[2,1],Y[2,3])
    - FUN can be any function involving two variables. By default, FUN = "*".
      - Please note outer(X,Y,"*") can be simplified as X %o% Y.
    - Example:

```
> x <- matrix(c(3:11),nrow = 3)
> y <- matrix(c(1,2,3,4),nrow = 2)
> outer(x,y)
, , 1, 1

     [,1] [,2] [,3]
[1,]    3    6    9
[2,]    4    7   10
[3,]    5    8   11

, , 2, 1

     [,1] [,2] [,3]
[1,]    6   12   18
[2,]    8   14   20
[3,]   10   16   22

, , 1, 2

     [,1] [,2] [,3]
[1,]    9   18   27
[2,]   12   21   30
[3,]   15   24   33

, , 2, 2

     [,1] [,2] [,3]
[1,]   12   24   36
[2,]   16   28   40
[3,]   20   32   44
```

Array Operations

✓ Applying functions

➡ Function *apply*() may be used to apply functions to dimensions or elements of an array

- Usage: Type *apply*[{array1}, MARGIN, FUN, …] at the command line to return a specific element.
  - MARGIN is a vector of subscripts or dimensions the function needs to be applied to. For example, in a matrix, rows have subscript 1 and columns, subscript 2.
- A mathematical function such as SQRT applied to a dimension will apply to every element
  - Example:

```
> x
     [,1] [,2] [,3]
[1,]    4   10   25
[2,]    7   11   14
> apply(x,1,sqrt)
         [,1]     [,2]
[1,] 2.000000 2.645751
[2,] 3.162278 3.316625
[3,] 5.000000 3.741657
> apply(x,2,sqrt)
         [,1]     [,2]     [,3]
[1,] 2.000000 3.162278 5.000000
[2,] 2.645751 3.316625 3.741657
```

- A statistical/arithmetic function such as MEAN applied to a dimension will produce one summary value
  - In the above example,

```
> apply(x,1,mean)
[1] 13.00000 10.66667
> apply(x,2,mean)
[1]  5.5 10.5 19.5
```

Exercise

Matrix Operations

- ✓ Subscripts of a matrix
  - ➡ A part of a matrix can be returned by using a subscript
    - Usage: Type *matrix*[{subscript1}, {subscript2}] at the command line to return a specific element. Subscript1, 2 refers to row, column positions respectively
    - Example: x[2,3] will return 6, where x = 1 2 3
4 5 6
7 8 9
  - ➡ To return a row, specify the row number
    - Usage: Type *matrix*[{rownum},] at the command line
    - Example: In the above example, x[1,] will return 1 2 3
  - ➡ To return a column, specify the column number
    - Usage: Type *matrix*[, {colnum}] at the command line
    - Example: In the above example, x[,2] will return 2 5 8
  - ➡ Methods used for vector subscripts also apply to matrices. These include
    - Specifying a sequence
    - Using a vector
    - Including a logical expression
    - Excluding elements with negative subscripts

Matrix Operations

✓ Subscripts of a matrix

➡ Index matrices may be used to specify and return part of a matrix

- In the below example, a 3 x 2 index matrix y is used to specify values in a 6 x 6 matrix x. Any value in x whose positions are defined in each row of y can be accessed using x[y]

```
> x <- matrix(1:36, nrow = 6)
> x
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    7   13   19   25   31
[2,]    2    8   14   20   26   32
[3,]    3    9   15   21   27   33
[4,]    4   10   16   22   28   34
[5,]    5   11   17   23   29   35
[6,]    6   12   18   24   30   36
> y <- matrix(c(3:5,2:4),nrow = 3)
> y
     [,1] [,2]
[1,]    3    2
[2,]    4    3
[3,]    5    4
> x[y] <- 50
> x
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    7   13   19   25   31
[2,]    2    8   14   20   26   32
[3,]    3   50   15   21   27   33
[4,]    4   10   50   22   28   34
[5,]    5   11   17   50   29   35
[6,]    6   12   18   24   30   36
>
```

- Max number of rows in y = number of elements in x
- Max number of columns in y = 2

Matrix Operations

✓ Diagonal matrix

➡ Function *diag*() can be used to create a diagonal matrix

- Usage: Type *diag*[{object}] at the command line
- Depending on the object, the function returns different results
- If the object is a vector of length n, it will return a n x n matrix where the diagonal contains the elements in the vector
- Example:

```
> y <- c(1,7,4)
> diag(y)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    7    0
[3,]    0    0    4
```

- If the object is an existing matrix, it will return the diagonal elements as a vector
- Example:

```
> D <- matrix(c(1:9),nrow=3)
> D
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> diag(D)
[1] 1 5 9
```

- If the object is a variable, say an integer of value n, it will return the n x n identity matrix
- Example:

```
> x <- 3
> diag(x)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
>
```

Matrix Operations

✓ Matrix Multiplication

➡ The operation %*% can be used to multiply two matrices

- The number of columns in the first matrix should be equal the number of rows of the second matrix
- The resulting matrix will have #rows = number of rows of matrix 1 and #columns = number of columns of matrix 2. For instance, if A is a 2 x 3 matrix and B is a 3 x 5 matrix, A %*% B will result in a 2 x 5 matrix
- Example:

```
> A <- matrix(c(1:12), nrow = 3)
> A
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> B <- matrix(c(13:36),ncol=6)
> B
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   13   17   21   25   29   33
[2,]   14   18   22   26   30   34
[3,]   15   19   23   27   31   35
[4,]   16   20   24   28   32   36
> C <- A %*% B
> C
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  334  422  510  598  686  774
[2,]  392  496  600  704  808  912
[3,]  450  570  690  810  930 1050
>
```

Matrix Operations

✓ Matrix Multiplication

➡ Function *crossprod*() may be used to get the matrix product of two matrices where the first matrix has been transposed

- Usage: Type *crossprod*({matrix1},{matrix2}) from the command line
- The resulting matrix will have #rows = number of columns of matrix 1 and #columns = number of columns of matrix 2. For instance, if A is a 2 x 3 matrix and B is a 2 x 5 matrix, *crossprod*(A,B) = A^T %*% B and will result in a 2 x 5 matrix
- Example:

```
> A <- matrix(c(1:6),nrow=2)
> A
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> B <- matrix(c(1:10),nrow=2)
> B
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> C <- crossprod(A,B)
> C
     [,1] [,2] [,3] [,4] [,5]
[1,]    5   11   17   23   29
[2,]   11   25   39   53   67
[3,]   17   39   61   83  105
>
```

➡ Function t*crossprod*() may be used to get the matrix product of two matrices where the second matrix has been transposed. It is identical to *crossprod*() except for the said detail

Matrix Operations

✓ Inverse of a matrix

➡ Function *solve*() may be used on an n x n square matrix to obtain its inverse

- Usage: Type *solve*({matrix}) from the command line
- The result will be an n x n matrix. If A^{-1} is the inverse of A, then A%*%A^{-1} = A^{-1}%*%A = I, where I is the n x n identity matrix
- Example:

```
> x <- matrix(c(2,2,3,2), nrow=2)
> x
     [,1] [,2]
[1,]    2    3
[2,]    2    2
> solve(x)
     [,1] [,2]
[1,]   -1  1.5
[2,]    1 -1.0
>
```

✓ Solving linear equations

➡ Function *solve*() may be used to solve linear equations. For an equation with n variables, if a %*% x = b represents the equation, the solution x = a^{-1}%*%b or *solve*(a,b).

- Usage: Type *solve*({matrix1}, {matrix2}) from the command line
- The result will be an n x 1 matrix.
- Example:

```
> a <- matrix(c(1,1,2,-1),nrow=2)
> b <- matrix(c(4,1),nrow=2)
> a
     [,1] [,2]
[1,]    1    2
[2,]    1   -1
> b
     [,1]
[1,]    4
[2,]    1
```

```
> x <- solve(a,b)
> x
     [,1]
[1,]    2
[2,]    1
>
```

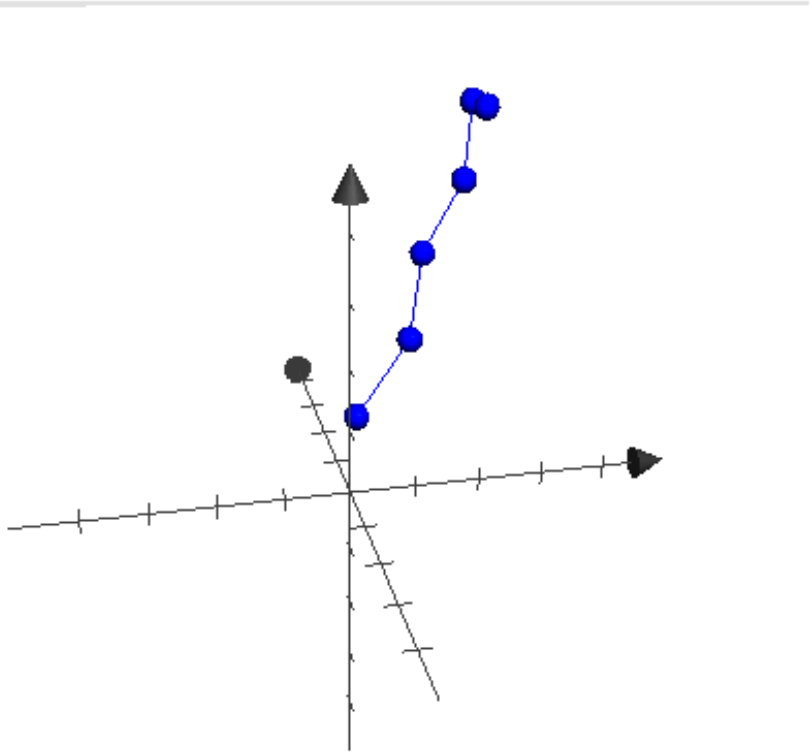
Matrix Operations

✓ Least Squares Regression

➡ Function *lsfit*() may be used to perform linear regression on a data stream and potential variables it may depend on

- Usage: Type *lsfit*({matrix of variables},{matrix of observations}, intercept =…, tolerance=…) from the command line
- Example: The following table/graph present data for linear regression

y	x1	x2
5	2.5	2.8
1	0.2	0.5
4	2.2	2.1
3	1.5	1.8
2	1.1	0.9
5	2.6	2.4



Matrix Operations

✓ Least Squares Regression

- Example (continued): *lsfit*(x,y) will yield the following:

```
$coefficients
Intercept        X1        X2        X3
0.2673910 1.0848048 0.7084881 0.0000000

$residuals
[1]  0.03683019  0.16140400 -0.14178668 -0.16987685 -0.09831561
[6]  0.21174495

$intercept
[1] TRUE

$qr
$qt
[1] -8.16496581  3.61264902  0.38888201 -0.22100357 -0.06525056
[6]  0.27887450
```

```
$qr
     Intercept         X2         X3            X1
[1,] -2.4494897 -4.1233077 -4.2866070 -2.449490e+00
[2,]  0.4082483  2.0852658  1.9062318  2.130096e-16
[3,]  0.4082483 -0.1342355  0.5488900  3.006887e-16
[4,]  0.4082483  0.2014531 -0.2325336  3.327645e-16
[5,]  0.4082483  0.3932752  0.7444122 -5.132936e-01
[6,]  0.4082483 -0.3260576  0.4968505  3.009664e-01

$qraux
[1] 1.408248 1.824875 1.380681 1.803709

$rank
[1] 3

$pivot
[1] 1 3 4 2

$tol
[1] 1e-07

attr(,"class")
[1] "qr"

Warning message:
In lsfit(x, y) : 'X' matrix was collinear
```

Exercise

Operations on Data Frames

✓ Accessing more than one column

➡ To access more than one column, use function *c*()

- Example: With data frame "*phones*", create another data frame that only lists phone name, price and OS

```
> ph <- phones[c("Price","OperSys")]
> ph
            Price     OperSys
iPhone        399         iOS
Galaxy        350     Android
Razr          200     Android
Pearl         399  Blackberry
Optimus One   299     Android
Lumia 800     299   Microsoft
```

✓ Accessing one or more rows

➡ To access multiple rows, use row numbers

- Example: To pull rows 2 to 4:

```
> phones[2:4,]
             Maker Price Country No_Sold     OperSys No_Apps  Carrier
Galaxy     Samsung   350   Korea  256121     Android 5716247  Verizon
Razr       Motorola  200     USA   26511     Android   12381   Sprint
Pearl   Blackberry   399  Canada  125819  Blackberry  123701   Rogers
```

➡ To access the first few, use function *head*()

- Usage: Type head({dataframe}, n) where n = no of rows
- Example:

```
> head(phones,2)
         Maker Price Country No_Sold  OperSys No_Apps  Carrier
iPhone   Apple   399     USA 2687161      iOS 3000000     AT&T
Galaxy Samsung   350   Korea  256121  Android 5716247  Verizon
```

Operations on Data Frames

✓ Accessing one or more rows

➡ To access the last few, use function *tail*()

- Usage: Type *tail*({dataframe}, n) where n = no of rows
- Example:

```
> tail(phones,2)
             Maker Price  Country No_Sold    OperSys No_Apps  Carrier
Optimus One     LG   299    Korea  123291    Android   12312     AT&T
Lumia 800    Nokia   299  Finland   23432  Microsoft   87699  Verizon
```

➡ To access multiple rows based on values in a column, use function *subset*()

- Usage: Type *subset*({object}, logical_expression,…) from the command line
- Example:

```
> subset(phones, Price > 300)
             Maker Price Country No_Sold     OperSys No_Apps  Carrier
iPhone       Apple   399     USA 2687161         iOS 3000000     AT&T
Galaxy     Samsung   350   Korea  256121     Android 5716247  Verizon
Pearl   Blackberry   399  Canada  125819  Blackberry  123701   Rogers
```

✓ Adding columns

➡ To add a column, use function *cbind*()

- Usage: Type *cbind*({dataframe1}, {vector1})
- Example:

```
> phones <- cbind(phones, Camera_Res = camera)
> phones
                Maker Price  Country No_Sold     OperSys No_Apps  Carrier   Camera_Res
iPhone          Apple   399      USA 2687161         iOS 3000000     AT&T 8 Megapixels
Galaxy        Samsung   350    Korea  256121     Android 5716247  Verizon 7 Megapixels
Razr          Motorola  200      USA   26511     Android   12381   Sprint 5 Megapixels
Pearl       Blackberry  399   Canada  125819  Blackberry  123701   Rogers 8 Megapixels
Optimus One         LG  299    Korea  123291     Android   12312     AT&T 8 Megapixels
Lumia 800       Nokia   299  Finland   23432   Microsoft   87699  Verizon 7 Megapixels
```

- *With this, the vector must have values in the same order as rows in the data frame*

Operations on Data Frames

✓ Combining two data frames

➡ To combine two data frames, use function *rbind*()

- Usage: Type *rbind*({dataframe1}, {dataframe2})
- Example:

```
> phones
                Maker Price  Country No_Sold     OperSys No_Apps  Carrier   Camera_Res
iPhone          Apple   399      USA 2687161         iOS 3000000     AT&T 8 Megapixels
Galaxy        Samsung   350    Korea  256121     Android 5716247  Verizon 7 Megapixels
Razr          Motorola  200      USA   26511     Android   12381   Sprint 5 Megapixels
Pearl       Blackberry  399   Canada  125819  Blackberry  123701   Rogers 8 Megapixels
Optimus One         LG  299    Korea  123291     Android   12312     AT&T 8 Megapixels
Lumia 800       Nokia   299  Finland   23432   Microsoft   87699  Verizon 7 Megapixels
> phones1
        Maker Price Country No_Sold  OperSys No_Apps  Carrier    Camera_Res
Xperia S  Sony   300   Japan   79792  Android  121211     AT&T  6 Megapixels
One X      HTC   250  Taiwan   99191  Android    1312  Verizon  6 Megapixels
> rbind(phones, phones1)
                Maker Price  Country No_Sold     OperSys No_Apps  Carrier    Camera_Res
iPhone          Apple   399      USA 2687161         iOS 3000000     AT&T  8 Megapixels
Galaxy        Samsung   350    Korea  256121     Android 5716247  Verizon  7 Megapixels
Razr          Motorola  200      USA   26511     Android   12381   Sprint  5 Megapixels
Pearl       Blackberry  399   Canada  125819  Blackberry  123701   Rogers  8 Megapixels
Optimus One         LG  299    Korea  123291     Android   12312     AT&T  8 Megapixels
Lumia 800       Nokia   299  Finland   23432   Microsoft   87699  Verizon  7 Megapixels
Xperia S         Sony   300    Japan   79792     Android  121211     AT&T  6 Megapixels
One X             HTC   250   Taiwan   99191     Android    1312  Verizon  6 Megapixels
```

Operations on Data Frames

✓ Combining two data frames

➡ In function *rbind*(), specific columns may be combined using function *c*()

- Usage: Type *rbind*({dataframe1 [*c*(column list)]}, {dataframe2 [*c*(column list)]})
- Example:

```
> phones
                Maker Price  Country No_Sold     OperSys No_Apps  Carrier   Camera_Res
iPhone          Apple   399      USA 2687161         iOS 3000000     AT&T 8 Megapixels
Galaxy        Samsung   350    Korea  256121     Android 5716247  Verizon 7 Megapixels
Razr          Motorola  200      USA   26511     Android   12381   Sprint 5 Megapixels
Pearl       Blackberry  399   Canada  125819  Blackberry  123701   Rogers 8 Megapixels
Optimus One         LG  299    Korea  123291     Android   12312     AT&T 8 Megapixels
Lumia 800       Nokia   299  Finland   23432   Microsoft   87699  Verizon 7 Megapixels
> phones1
        Maker Price Country No_Sold  OperSys No_Apps  Carrier    Camera_Res
Xperia S  Sony   300   Japan   79792  Android  121211     AT&T  6 Megapixels
One X      HTC   250  Taiwan   99191  Android    1312  Verizon  6 Megapixels
> rbind(phones[c("Maker","Price")],phones1[c("Maker","Price")])
                Maker Price
iPhone          Apple   399
Galaxy        Samsung   350
Razr          Motorola  200
Pearl       Blackberry  399
Optimus One         LG  299
Lumia 800       Nokia   299
Xperia S         Sony   300
One X             HTC   250
```

- In both examples, the result of *rbind*() may be assigned to an object

Operations on Data Frames

✓ Obtaining summaries

➡ Function *aggregate*() may be used to obtain summary level info from a data frame

- Usage: Type *aggregate*({dataframe1}, by, FUN,…), where *by* is the column used to aggregate data by, and FUN is the summary function
- Example: In the *phones* data set, obtain mean Price, sales volume and # of apps by Operating System

```
> phones
                Maker Price  Country No_Sold     OperSys No_Apps  Carrier   Camera_Res
iPhone          Apple   399      USA 2687161         iOS 3000000     AT&T 8 Megapixels
Galaxy        Samsung   350    Korea  256121     Android 5716247  Verizon 7 Megapixels
Razr          Motorola  200      USA   26511     Android   12381   Sprint 5 Megapixels
Pearl       Blackberry  399   Canada  125819  Blackberry  123701   Rogers 8 Megapixels
Optimus One         LG  299    Korea  123291     Android   12312     AT&T 8 Megapixels
Lumia 800       Nokia   299  Finland   23432   Microsoft   87699  Verizon 7 Megapixels
> aggregate(phones,list(phones$OperSys),mean)
      Group.1 Maker Price Country   No_Sold OperSys No_Apps Carrier Camera_Res
1     Android    NA   283      NA  135307.7      NA 1913647      NA         NA
2  Blackberry    NA   399      NA  125819.0      NA  123701      NA         NA
3   Microsoft    NA   299      NA   23432.0      NA   87699      NA         NA
4         iOS    NA   399      NA 2687161.0      NA 3000000      NA         NA
There were 20 warnings (use warnings() to see them)
```

- With this approach, all other text columns are still retained in the result and returned as *NA*

Operations on Data Frames

✓ Changing column values

➡ Function *tranform*() may be used to modify a column or a set of columns

- Usage: Type *transform*({dataframe}, equation)
- Example: In the *phones* data set, to convert Price from USD to CAD (using a factor 1.1):

```
> transform(phones,Price=Price*1.1)
                Maker Price  Country No_Sold     OperSys No_Apps  Carrier
iPhone          Apple 438.9      USA 2687161         iOS 3000000     AT&T
Galaxy        Samsung 385.0    Korea  256121     Android 5716247  Verizon
Razr         Motorola 220.0      USA   26511     Android   12381   Sprint
Pearl     Blackberry 438.9   Canada  125819  Blackberry  123701   Rogers
Optimus One       LG 328.9    Korea  123291     Android   12312     AT&T
Lumia 800     Nokia 328.9  Finland   23432   Microsoft   87699  Verizon
```

Exercise

Operations on Factors

✓ Applying functions

➡ Function *tapply*() allows for a function to be applied over an object at the levels of a factor. The object and the factor have the same length

- Usage: Type *tapply*({Object}, {Factor}, Function = , …) from the command line. {Object} is a numeric object with the same length as {Factor}. Function can be any means of summarizing {Object}
- Example: In table *infert* in package *datasets*, evaluate the average age of women tested at various levels of education

```
> infert
    education age parity induced case spontaneous stratum pooled.stratum
1      0-5yrs  26      6       1    1           2       1              3
2      0-5yrs  42      1       1    1           0       2              1
3      0-5yrs  39      6       2    1           0       3              4
4      0-5yrs  34      4       2    1           0       4              2
                ⋮
245   12+ yrs  34      1       0    0           0      80             47
246   12+ yrs  35      2       2    0           0      81             54
247   12+ yrs  29      1       0    0           1      82             43
248   12+ yrs  23      1       0    0           1      83             40
> tapply(infert$age, infert$education, mean)
  0-5yrs  6-11yrs  12+ yrs
35.25000 32.85000 29.72414
```

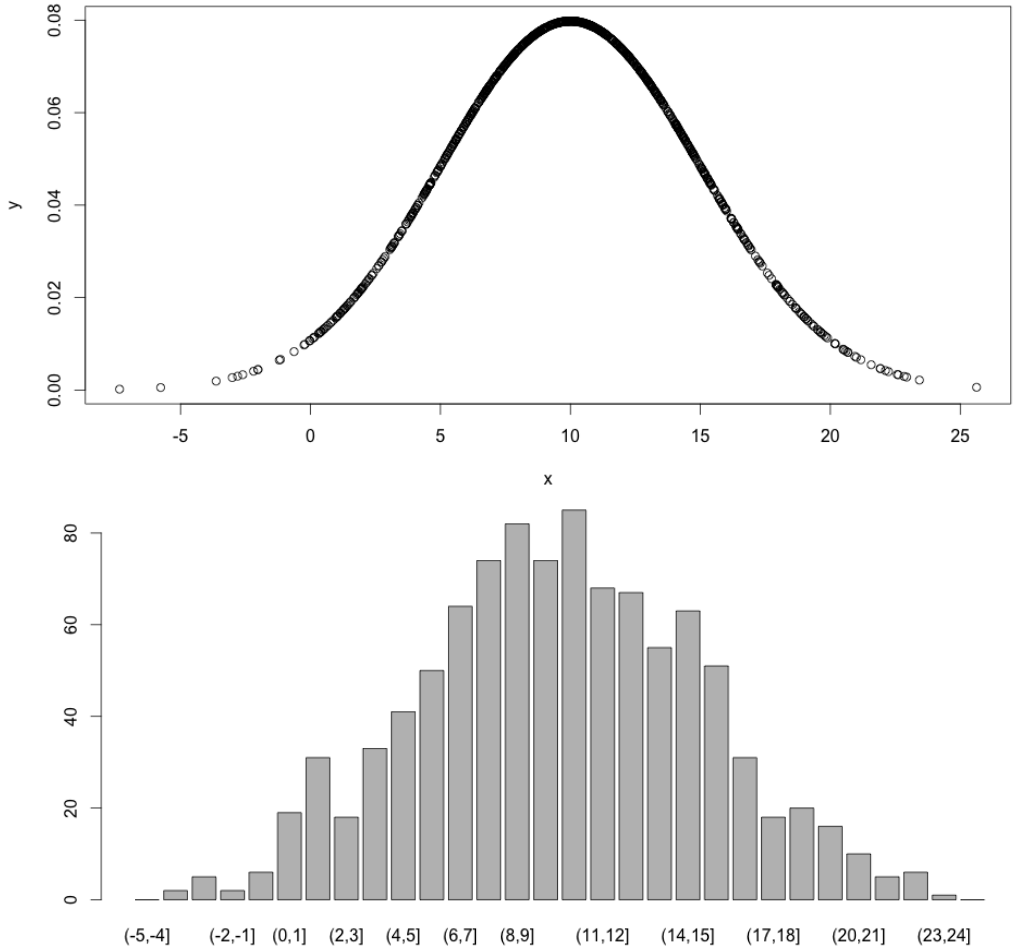
Operations on Factors

✓ Creating levels from numeric data

➡ Function *cut*() creates levels in numeric data by dividing the range of the data into equal intervals and coding each level by the data that fall in that interval

- Usage: Type *cut*({Object}, breaks = ,…) from the command line. Parameter *breaks* specifies the intervals created
- Example:

```
> x <- rnorm(1000,10,5)
> y <- dnorm(x,10,5)
> plot(x,y)
> z <- cut(x,c(-5:25))
> plot(z)
```



Operations on Factors

✓ Generating factors from patterns

➡ Function *gl*() generates factors by specifying levels and their patterns

- Usage: Type *gl*(n, k, length = , labels = , …) from the command line. Parameter *n* specifies the # of levels, parameter *k* specifies # of replicas, *length* = n*k, and *labels* is an optional vector of labels
- Example:

```
> w <- gl(3,4,labels=c("Ann","Rich","Baker"))
> w
 [1] Ann   Ann   Ann   Ann   Rich  Rich  Rich  Rich  Baker Baker
[11] Baker Baker
Levels: Ann Rich Baker
> w <- gl(3,4)
> w
 [1] 1 1 1 1 2 2 2 2 3 3 3 3
Levels: 1 2 3
```

Operations on Factors

✓ Re-ordering levels

➡ To score or weight the levels of a factor based on other data, use function *reorder*()

- Usage: Type *reorder*({Factor},{Object}, Function = ,…) from the command line. {Object} is a numeric object with the same length as {Factor}. Function can be any means of scoring {Factor}
- Example: Using Edgar Anderson's iris data

```
> attach(iris)
The following object(s) are masked from 'iris (position 3)':

    Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species
> z <- reorder(Species,Sepal.Length,mean)
> z
  [1] setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa
 [10] setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa
 [19] setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa     setosa
                                    ⋮
[127] virginica  virginica  virginica  virginica  virginica  virginica  virginica  virginica  virginica
[136] virginica  virginica  virginica  virginica  virginica  virginica  virginica  virginica  virginica
[145] virginica  virginica  virginica  virginica  virginica  virginica
attr(,"scores")
    setosa versicolor  virginica
     5.006      5.936      6.588
Levels: setosa versicolor virginica
> y <- attr(z,"scores")
> y
    setosa versicolor  virginica
     5.006      5.936      6.588
```

Operations on Factors

✓ Returning the index positions

➡ Function *unclass*() returns the index position of each value in the factor, based on its level

- Usage: Type *unclass*({Factor}, …) from the command line.
- Example:

```
> x <- as.factor(rep(c(1:3)*7,10))
> x
 [1] 7  14 21 7  14 21 7  14 21 7  14 21 7  14 21 7  14 21 7  14 21 7  14 21 7  14 21
Levels: 7 14 21
> unclass(x)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
attr(,"levels")
[1] "7"  "14" "21"
```

Exercise

Operations on Text data

✓ Length of a string

➡ Function *nchar*() is used to obtain the length of a string

- Usage: Type *nchar*({object}) from the command prompt
- Example:

```
> x <- "This is a test"
> nchar(x)
[1] 14
```

✓ Parts of a string

➡ Function *substr*() is used to obtain part of a string

- Usage: Type *substr*({object}, start, stop) from the command prompt
- Example:

```
> x <- "This is a test"
> substr(x,6,9)
[1] "is a"
```

➡ *substr*() can also be used to replace part of a string by using an assignment

- Example:

```
> x <- "Price = XXXX10"
> x
[1] "Price = XXXX10"
> substr(x,9,12) <- "USD "
> x
[1] "Price = USD 10"
```

✓ Concatenating strings

➡ Function *paste*() is used to combine multiple strings

- Usage: Type *paste*({string1}, …, sep =) from the command prompt. By default, sep = " ".
- Example:

```
> x <- "This is a town."
> y <- "Its name is melancholy"
> paste(x,y)
[1] "This is a town. Its name is melancholy"
```

Operations on Text data

- ✓ Pattern matching
  - ➡ Function *grep*() is used to search for a string in another string or a character vector
    - Usage: Type *grep*({pattern}, {object}, value = , ignore.case = , fixed =) from the command prompt
    - Examples:

```
> x
[1] "this"  "IS"    "tom"   "hiss"  "peter" "ron"   "miss"
> grep("is",x,value = TRUE)
[1] "this" "hiss" "miss"
> grep("is",x,value = FALSE)
[1] 1 4 7
> grep("is",x,value = TRUE, ignore.case=TRUE)
[1] "this" "IS"   "hiss" "miss"
> grep("is",x,value = TRUE, ignore.case=TRUE, fixed=TRUE)
[1] "this" "hiss" "miss"
Warning message:
In grep("is", x, value = TRUE, ignore.case = TRUE, fixed = TRUE) :
  argument 'ignore.case = TRUE' will be ignored
```

    - Functions *grep l* (), *regexpr*(), *gregexpr*(), *regexec*() are variations

- ✓ Built in constants
  - ➡ Constant *letters* contains the English alphabet in lowercase
    - Usage: Type *letters* to return a vector containing the alphabet in
    - Examples:

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x"
[25] "y" "z"
> letters[4]
[1] "d"
```

  - ➡ Constant *LETTERS* contains the English alphabet in uppercase

Operations on Text data

✓ Replacement

➡ Function *gsub*() is used to search for a string in another string or a character vector

- Usage: Type *gsub*({pattern}, {replacement}, {object}, ignore.case = , fixed =) from the command prompt
- Examples:

```
> x
[1] "this"  "IS"    "tom"   "hiss"  "peter" "ron"   "miss"
> gsub("is","at",x,ignore.case=TRUE)
[1] "that"  "at"    "tom"   "hats"  "peter" "ron"   "mats"
> gsub("is","ATHER",x,ignore.case=FALSE)
[1] "thATHER" "IS"      "tom"     "hATHERs" "peter"   "ron"     "mATHERs"
> gsub("is","ATHER",x,ignore.case=TRUE)
[1] "thATHER" "ATHER"   "tom"     "hATHERs" "peter"   "ron"     "mATHERs"
```

➡ Function *sub*() is identical to *gsub*(), except that it only replaces the first occurrence of the pattern

➡ Using () to specify a text group and backreference \1 provides added sophistication in finding pattern and replacement

- Examples:

```
> x <- c("This is a test","Of capturing groups", "and backreferences")
> sub("(a+)","D\\1D",x)
[1] "This is DaD test"      "Of cDaDpturing groups" "DaDnd backreferences"
> gsub("(a+)","D\\1D",x)
[1] "This is DaD test"       "Of cDaDpturing groups"  "DaDnd bDaDckreferences"
```

Exercise

Operations on Dates

- Function *as.Date*() is used to create a Date object
  - Usage: Type *as.Date*({object}, ...) to create a Date object
  - Example: x <- *as.Date*("09-04-2012", "%d-%m-%Y") will result in x = "2012-04-09"
  - Usage: Type *as.Date*({object}, *format* = "") to specify a format for the input
  - Example: x <- *as.Date*("09-04-2012", "%m-%d-%Y") will result in x = "2012-09-04"
  - Usage: Type *as.Date*({Number}, *origin* = "") to calculate a date as the number of days from an origin
  - Example: x <- *as.Date*(30, origin = "2012-04-01") will result in x = "2012-05-01"

- Formatting dates
  - Dates follow default formats set by ISO 8601: "YYYY-MM-DD"
  - To assign a format to the *Date* object, use function *format*()
  - Usage: Type *format*({object}, …) to convert to MM-DD-YYYY
  - Example: x <- format("2012-04-01", "%m-%d-%Y") will result in x = "04-01-2012"

- Arithmetic
  - If a number is added to or subtracted from a date, R modifies the date by the said number of days
  - Example: x + 31 where x = "2012-05-01" will result in "2012-06-01"

- System date and Time
  - Functions Sys.Date() and Sys.time() are used to return the system date and time respectively
  - Usage: Type *Sys.Date*() to return the system date or type Sys.time() for the time

Operations on Dates

- ✓ POSIX time
  - ➡ The POSIXct and POSIXlt classes are used to work with POSIX times in R
  - ➡ Function *as.POSIXct*() is used to store dates as the number of seconds since 1/1/1970
    - Usage: Type *as.POSIXct*({object}, ...) to create a POSIXct object
    - Example: x <- *as.POSIXct*("2012-09-04") will result in x = "2012-04-09 EDT"
    - Example: x <- *as.POSIXct*("2012-09-04 07:29:04") will result in x = "2012-09-04 07:29:04 EDT"
  - ➡ Function *as.POSIXlt*() is used to store dates as a List
    - Usage: Type *as.POSIXlt*({object}, ...) to create a POSIXlt object
    - Example: x <- *as.POSIXlt*("2012-09-04") will result in x = "2012-09-04" with type List
  - ➡ Function *strptime*() is used to create a POSIXlt object from individual strings that represents parts of a date. Allows for different date formats
    - Usage: Type *strptime*({object}, format) to create a POSIXlt object
    - Example:
      - dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
      - times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03", "16:56:26")
      - x <- paste(dates, times)
      - y <- strptime(x, "%m/%d/%y %H:%M:%S")
      - Results in y =
      - [1] "1992-02-27 23:03:20" "1992-02-27 22:29:56" "1992-01-14 01:03:30"
      - [4] "1992-02-28 18:21:03" "1992-02-01 16:56:26"

Operations on Dates

- ✓ POSIX time
  - ➡ Functions *weekdays*(), and months() may be used to extract the week day or month from a POSIX date object
    - Usage: Type *weekdays*({object}, abbreviate) or *months*({object}, abbreviate). Parameter *abbreviate*, if TRUE will provide abbreviations of the names
    - Example:

```
> y <- as.POSIXlt("2012-09-24")
> y
[1] "2012-09-24"
> months(y)
[1] "September"
> weekdays(y)
[1] "Monday"
```

- ✓ Built in constants
  - ➡ Constant *month.abb* contains the 3 letter abbreviations of months
    - Usage: Type *month.abb* to return a vector containing the english alphabet in
    - Examples:

```
> month.abb
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
> month.abb[4]
[1] "Apr"
```

  - ➡ Constant *month.name* contains the month names
    - Usage: Type *month.name* to return a vector containing the english alphabet in
    - Examples:

```
> month.name
 [1] "January"   "February"  "March"     "April"     "May"       "June"      "July"      "August"
 [9] "September" "October"   "November"  "December"
> month.name[c(4:6)]
[1] "April" "May"   "June"
```

Exercise