
Managing epidemiologic data in R

Tomás Aragón
Wayne Enanoria

September 18, 2007

3.1 Entering and importing data

There are many ways of getting your data into R for analysis. In the section that follows we review how to enter the University Group Diabetes Program data (Table 3.1) as well as the original data from a comma-delimited text file. We will use the following approaches:

- Entering data at the command prompt
- Importing data from a file
- Importing data using an URL

3.1.1 Entering data at the command prompt

We review four methods. For Methods 1 and 2, data are entered directly at the command prompt. Using Method 3, data is entered into a text editor (using Method 1 or 2) and then pasted into R or run as a batch file. And, for Method 4 we use R's spreadsheet editor.

Table 3.1. Deaths among subjects who received tolbutamide and placebo in the University Group Diabetes Program (1970), stratifying by age

	Age<55		Age≥55		Combined	
	Tolbutamide	Placebo	Tolbutamide	Placebo	Tolbutamide	Placebo
Deaths	8	5	22	16	30	21
Survivors	98	115	76	69	174	184

Method 1

For review, a convenient way to enter data at the command prompt is to use the `c` function:

```

> #enter data for a vector
> vec1 <- c(8, 98, 5, 115); vec1
[1] 8 98 5 115
> vec2 <- c(22, 76, 16, 69); vec2
[1] 22 76 16 69
>
> #enter data for a matrix
> mtx1 <- matrix(vec1, 2, 2); mtx1
      [,1] [,2]
[1,]    8    5
[2,]   98  115
> mtx2 <- matrix(vec2, 2, 2); mtx2
      [,1] [,2]
[1,]   22  16
[2,]   76  69
>
> #enter data for an array
> udat <- array(c(vec1, vec2), c(2, 2, 2)); udat
, , 1
      [,1] [,2]
[1,]    8    5
[2,]   98  115
, , 2
      [,1] [,2]
[1,]   22  16
[2,]   76  69

> udat.tot <- apply(udat, c(1, 2), sum); udat.tot
      [,1] [,2]
[1,]   30  21
[2,]  174 184
>
> #enter a list
> x <- list(crude.data = udat.tot, stratified.data = udat)
> x$crude.data
      [,1] [,2]
[1,]   30  21
[2,]  174 184

```

```

> x$stratified
, , 1

      [,1] [,2]
[1,]    8    5
[2,]   98  115

, , 2

      [,1] [,2]
[1,]   22   16
[2,]   76   69

>
> #enter simple data frame
> subjname <- c("Pedro", "Paulo", "Maria")
> subjno <- 1:length(subjname)
> age <- c(34, 56, 56)
> sex <- c("Male", "Male", "Female")
> dat <- data.frame(subjno, subjname, age, sex); dat
  subjno subjname age  sex
1      1   Pedro  34  Male
2      2   Paulo  56  Male
3      3   Maria  56 Female
>
> #enter a simple function
> odds.ratio <- function(aa, bb, cc, dd){ aa*dd / (bb*cc)}
> odds.ratio(30, 174, 21, 184)
[1] 1.510673

```

Method 2

Method 2 is identical to Method 1 except one uses the scan function. It does not matter if you enter the numbers on different lines, it will still be a vector. Remember that you must press the Enter key twice after you have entered the last number.

```

> udat.tot <- scan()
1: 30 174
3: 21 184
5:
Read 4 items
> udat.tot
[1] 30 174 21 184

```

To read in a matrix at the command prompt combine the `matrix` and `scan` functions. Again, it does not matter on what lines you enter the data, as long as they are in the correct order because the `matrix` function reads data in column-wise.

```
> udat.tot <- matrix(scan(), 2, 2)
1: 30 174 21 184
5:
Read 4 items
> udat.tot
      [,1] [,2]
[1,]   30   21
[2,]  174  184

> udat.tot <- matrix(scan(), 2, 2, byrow=T) #read data row-wise
1: 30 21 174 184
5:
Read 4 items
> udat.tot
      [,1] [,2]
[1,]   30   21
[2,]  174  184
```

To read in an array at the command prompt combine the `array` and `scan` functions. Again, it does not matter on what lines you enter the data, as long as they are in the correct order because the `array` function reads the numbers column-wise. In this example we include the `dimnames` argument.

```
> udat <- array(scan(), dim = c(2, 2, 2),
+   dimnames = list(Vital.Status = c("Dead", "Survived"),
+   Treatment = c("Tolbutamide", "Placebo"),
+   Age.Group = c("<55", "55+")))
1: 8 98 5 115 22 76 16 69
9:
Read 8 items
> udat
, , Age.Group = <55

      Treatment
Vital.Status Tolbutamide Placebo
Dead          8          5
Survived      98         115

, , Age.Group = 55+

      Treatment
```

Vital.Status	Tolbutamide	Placebo
Dead	22	16
Survived	76	69

To read in a list of vectors of the same length (“fields”) at the command prompt combine the `list` and `scan` function. Notice that you will need to specify the type of data that will go into each “bin” or “field.” This is done by specifying the `what` argument as a list. This list must be values that are either logical, integer, numeric, or character. For example, for a character vector you can use any expression, say x , that would evaluate to `TRUE` for `is.character(x)`. For brevity, use `"` for character, `0` for numeric, `1:2` for integer, and `T` or `F` for logical. Look at this example:

```
> dat <- scan("", what = list(1:2, "", 0, "", T))
1: 3 "John Paul" 84.5 Male F
2: 4 "Jane Doe" 34.5 Female T
3:
Read 2 records
> str(dat)
List of 5
 $ : int [1:2] 3 4
 $ : chr [1:2] "John Paul" "Jane Doe"
 $ : num [1:2] 84.5 34.5
 $ : chr [1:2] "Male" "Female"
 $ : logi [1:2] FALSE TRUE
>
> #same example with field names
> dat <- scan("", what = list(id=1:2, name="", age=0, sex="",
+   dead=TRUE))
1: 3 "John Paul" 84.5 Male F
2: 4 "Jane Doe" 34.5 Female T
3:
Read 2 records
> str(dat)
List of 5
 $ id : int [1:2] 3 4
 $ name: chr [1:2] "John Paul" "Jane Doe"
 $ age : num [1:2] 84.5 34.5
 $ sex : chr [1:2] "Male" "Female"
 $ dead: logi [1:2] FALSE TRUE
```

To read in a data frame at the command prompt combine the `data.frame`, `scan`, and `list` functions.

```
> dat <- data.frame(scan("", what = list(id=1:2, name="",
+   age=0, sex="", dead=T)) )
```

```

1: 3 "John Paul" 84.5 Male F
2: 4 "Jane Doe" 34.5 Female T
3:
Read 2 records
> dat
  id      name age  sex  dead
1  3 John Paul 84.5  Male FALSE
2  4 Jane Doe 34.5 Female TRUE

```

Method 3

In Method 3, data is entered into a text editor (using Method 1 or 2) and then pasted into R or run as a batch file using the `source` function. For example, the following code is in a text editor (such as Notepad in Windows) and saved to a file named `job01.R`.

```

x <- 1:10
x

```

One can copy and paste this code into R at the command prompt.

```

> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10

```

However, if you run the code using the `source` function, it will only display to the screen those objects that are printed using the `print` function. Here is the text editor code again, but including `print`.

```

x <- 1:10
print(x)

```

Now, run the program file (`job01.R`) using `source` at the command prompt.

```

> source("c:/job01.R")
[1] 1 2 3 4 5 6 7 8 9 10

```

In general, we highly recommend using a text editor for all your work. The program file (e.g., `job01.R`) created with the text editor facilitates documenting your code, reviewing your code, debugging your code, replicating your analytic steps, and auditing by external reviewers.

Method 4

Method 4 uses R's spreadsheet editor. This is not a preferred method because we like the original data to be in a text editor or read in from a data file. We will be using the `data.entry` and `edit` functions. The `data.entry` function allows editing of an existing object and automatically saving the changes to the original object name. In contrast, the `edit` function allows editing of an

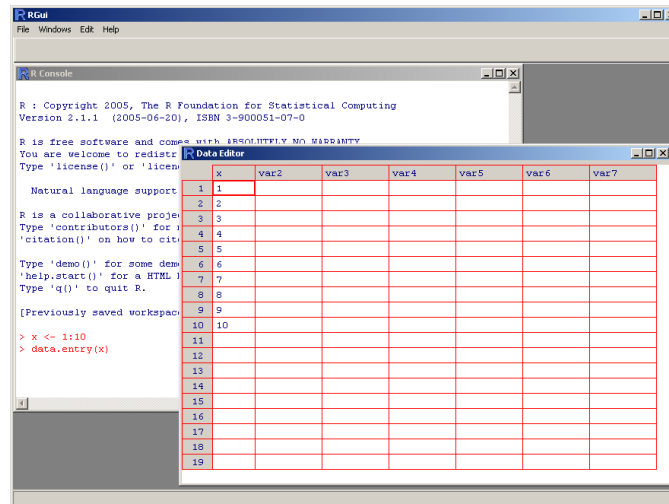


Fig. 3.1. Select Help from the main menu.

existing object but it will not save the changes to the original object name; you must explicitly assign it to an object name (even if it is the original name).

To enter a vector you need to initialize a vector and then use the `data.entry` function (Figure 3.1).

```

> x <- numeric(10) #Initialize vector with zeros
> x
[1] 0 0 0 0 0 0 0 0 0 0
> data.entry(x) #Enter numbers, then close window
> x
[1] 1 2 3 4 5 6 7 8 9 10

```

However, the `edit` function applied to a vector does not open a spreadsheet. Try the `edit` function and see what happens.

```

xnew <- edit(numeric(10)) #Edit number, then close window

```

To enter data into a spreadsheet matrix first initialize a matrix and then use the `data.entry` or `edit` function. Notice that the editor added default column names. However, to add your own column names just click on the column heading with your mouse pointer (unfortunately you cannot give row names).

```

> xnew <- matrix(numeric(4),2,2)
> data.entry(xnew)
> xnew <- edit(xnew) #equivalent
>
> #open spreadsheet editor in one step

```

```
> xnew <- edit(matrix(numeric(4),2,2))
> xnew
      col1 col2
[1,]  11   33
[2,]  22   44
```

Arrays and nontabular lists cannot be entered using a spreadsheet editor. Hence, we begin to see the limitations of spreadsheet-type approach to data analysis. One type of list, the data frame, can be entered using the `edit` function.

To enter a data frame use the `edit` function. However, you do not need to initialize a data frame (unlike with a matrix). Again, click on the column headings to enter column names.

```
> df <- edit(data.frame()) #Spreadsheet screen not shown
> df
      mykids age
1 Tomasito   7
2 Luisito    6
3 Angelita   3
```

When using the `edit` function to create a new data frame you must assign it an object name to save the data frame. Later we will see that when we edit an existing data object we can use the `edit` or `fix` function. The `fix` function differs in that `fix(data_object)` saves your edits directly back to `data_object` without the need to make a new assignment.

```
mypower <- function(x, n){x^n}
fix(mypower)
mypower <- edit(mypower) #equivalent
```

3.1.2 Importing data from a file

Reading an ASCII text data file

In this section we review how to read the following types of text data files:

- Comma-separated variable (csv) data file (\pm headers and \pm row names)
- Fixed width formatted data file (\pm headers and \pm row names)

Here is the University Group Diabetes Program randomized clinical trial text data file that is comma-delimited, and includes row names and a header (ugdp.txt)¹. The header is the first line that contains the column (field) names. The row names is the first column that starts on the second line and uniquely identifies each row. Notice that the row names does not have a column name associated with it. A data file can come with either row names or header,

¹ Available at <http://www.medepi.net/data/ugdp.txt>

neither, or both. Our preference is to work with data files that have a header and data values that are self-explanatory. Even without a data dictionary one can still make sense out of this data set.

```
Status,Treatment,Agegrp
1,Dead,Tolbutamide,<55
2,Dead,Tolbutamide,<55
...
408,Survived,Placebo,55+
409,Survived,Placebo,55+
```

Notice that the header row has 3 items, and the second row has 4 items. That's because the row names starts with the second row and has no column name. This data file can be read in using the `read.table` function.

```
> ud <- read.table("http://www.medepi.net/data/ugdp.txt",
+   header = TRUE, sep = ",")
> head(ud) #displays 1st 6 lines
  Status Treatment Agegrp
1  Dead Tolbutamide  <55
2  Dead Tolbutamide  <55
3  Dead Tolbutamide  <55
4  Dead Tolbutamide  <55
5  Dead Tolbutamide  <55
6  Dead Tolbutamide  <55
```

Here is the same data file as it would appear without row names and without a header (ugdp2.txt).

```
Dead,Tolbutamide,<55
Dead,Tolbutamide,<55
...
Survived,Placebo,55+
Survived,Placebo,55+
```

This data file can be read in using the `read.table` function.

```
> cnames <- c("Status", "Treatment", "Agegrp")
> udat2 <- read.table("http://www.medepi.net/data/ugdp2.txt",
+   header = FALSE, sep = ",", col.names = cnames)
> head(udat2)
  Status Treatment Agegrp
1  Dead Tolbutamide  <55
2  Dead Tolbutamide  <55
3  Dead Tolbutamide  <55
4  Dead Tolbutamide  <55
5  Dead Tolbutamide  <55
6  Dead Tolbutamide  <55
```

Here is the same data file as it might appear as a fix formatted file. In this file, columns 1 to 8 are for field #1, columns 9 to 19 are for field #2, and columns 20 to 22 are for field #3. This type of data file is more compact. One needs a data dictionary to know which columns contain which fields.

```
Dead    Tolbutamide<55
Dead    Tolbutamide<55
...
SurvivedPlacebo  55+
SurvivedPlacebo  55+
```

Here is how this data file would be read in using the `read.fwf` function.

```
> cnames <- c("Status", "Treatment", "Agegrp")
> udat3 <- read.fwf("http://www.medepi.net/data/ugdp3.txt",
+   width = c(8, 11, 3), col.names = cnames, strip.white=TRUE)
> head(udat3)
  Status Treatment Agegrp
1  Dead Tolbutamide  <55
2  Dead Tolbutamide  <55
3  Dead Tolbutamide  <55
4  Dead Tolbutamide  <55
5  Dead Tolbutamide  <55
6  Dead Tolbutamide  <55
```

Finally, here is the same data file as it might appear as a fixed width formatted file but with numeric codes (ugdp4.txt). In this file, column 1 is for field #1, column 2 is for field #2, and column 3 is for field #3. This type of text data file is the most compact, however, one needs a data dictionary to make sense of all the 1s and 2s.

```
121
121
...
212
212
```

Here is how this data file would be read in using the `read.fwf` function.

```
> cnames <- c("Status", "Treatment", "Agegrp")
> udat4 <- read.fwf("http://www.medepi.net/data/ugdp4.txt",
+   width = c(1, 1, 1), col.names = cnames)
> head(udat4)
  Status Treatment Agegrp
1     1         2     1
2     1         2     1
3     1         2     1
4     1         2     1
5     1         2     1
```

```
6      1      2      1
```

R has other functions for reading text data files (`read.csv`, `read.csv2`, `read.delim`, `read.delim2`). In general, `read.table` is the function used most commonly for reading in data files.

Reading data from a binary format (e.g., Stata, Epi Info)

To read data that comes in a binary or proprietary format load the `foreign` package using the `library` function. To review available functions in the the `foreign` package try `help(package = foreign)`. For example, here we read in the ‘infert’ data set which is also available as a Stata data file².

```
> idat <- read.dta("c:/.../data/infert.dta")
> head(idat)[,1:8]
  id education age parity induced case spontaneous stratum
1  1         0  26     6        1    1           2        1
2  2         0  42     1        1    1           0        2
3  3         0  39     6        2    1           0        3
4  4         0  34     4        2    1           0        4
5  5         1  35     3        1    1           1        5
6  6         1  36     4        2    1           1        6
```

3.1.3 Importing data using a URL

As we have already seen, text data files can be read directly off a web server into R using the `read.table` function. Here we load the Western Collaborative Group Study data directly off a web server.

```
> wdat <- read.table("http://www.medepi.net/data/wcgs.txt",
+   header = TRUE, sep = ",")
> str(wdat)
'data.frame':  3154 obs. of  14 variables:
 $ id      : int  2001 2002 2003 2004 2005 2006 2007 2010 ...
 $ age0    : int  49 42 42 41 59 44 44 40 43 42 ...
 $ height0 : int  73 70 69 68 70 72 72 71 72 70 ...
 $ weight0 : int  150 160 160 152 150 204 164 150 190 175 ...
 $ sbp0    : int  110 154 110 124 144 150 130 138 146 132 ...
 $ dbp0    : int  76 84 78 78 86 90 84 60 76 90 ...
 $ chol0   : int  225 177 181 132 255 182 155 140 149 325 ...
 $ behpat0 : int  2 2 3 4 3 4 4 2 3 2 ...
 $ ncigs0  : int  25 20 0 20 20 0 0 0 25 0 ...
 $ dibpat0 : int  1 1 0 0 0 0 0 1 0 1 ...
 $ chd69   : int  0 0 0 0 1 0 0 0 0 0 ...
```

² Available at <http://www.medepi.net/data/infert.dta>

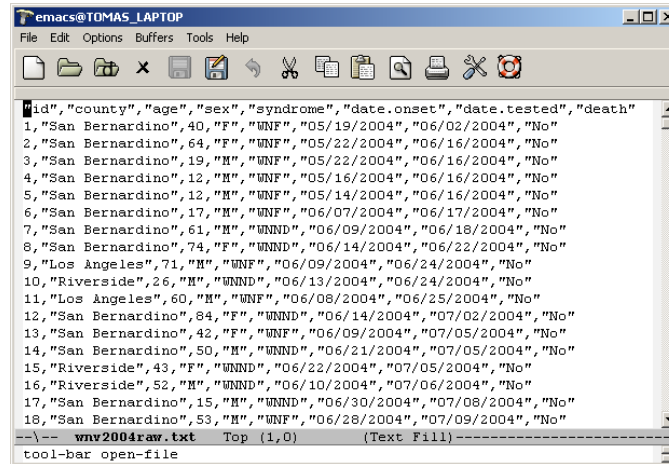


Fig. 3.2. Editing West Nile virus human surveillance data in text editor. Source: California Department of Health Services, 2004

```
$ typechd: int  0 0 0 0 1 0 0 0 0 0 ...
$ time169: int 1664 3071 3071 3064 1885 3102 3074 1032 ...
$ arcus0 : int  0 1 0 0 1 0 0 0 0 1 ...
```

3.2 Editing data

In the ideal setting, your data has already been checked, errors corrected, and ready to be analyzed. Post-collection data editing can be minimized by good design and data collection. However, you may still need to make corrections or changes in data values.

3.2.1 Text editor

For small data sets, it may be convenient to edit the data in your favorite text editor. Key-recording macros, and search and replace tools can be very useful and efficient. Figure 3.2 displays West Nile virus (WNV) infection surveillance data³. This file is a comma-delimited data file with a header.

3.2.2 The `data.entry`, `edit`, or `fix` functions

For vector and matrices you can use the `data.entry` function to edit these data object elements. For data frames and functions use the `edit` or `fix` functions. Remember that changes made with the `edit` function are not saved

³ Raw data set available at <http://www.medepi.net/data/wnv2004raw.txt>, and clean data set available at <http://www.medepi.net/data/wnv2004fin.txt>

	id	county	age	sex	syndrome	date.onset	date.tested	death
1	1	San Bernardino	40	F	WNF	05/19/2004	06/02/2004	No
2	2	San Bernardino	64	F	WNF	05/22/2004	06/16/2004	No
3	3	San Bernardino	19	M	WNF	05/22/2004	06/16/2004	No
4	4	San Bernardino	12	M	WNF	05/16/2004	06/16/2004	No
5	5	San Bernardino	12	M	WNF	05/14/2004	06/16/2004	No
6	6	San Bernardino	17	M	WNF	06/07/2004	06/17/2004	No
7	7	San Bernardino	61	M	WNND	06/09/2004	06/18/2004	No
8	8	San Bernardino	74	F	WNND	06/14/2004	06/22/2004	No
9	9	Los Angeles	71	M	WNF	06/09/2004	06/24/2004	No
10	10	Riverside	26	M	WNND	06/13/2004	06/24/2004	No
11	11	Los Angeles	60	M	WNF	06/08/2004	06/25/2004	No
12	12	San Bernardino	84	F	WNND	06/14/2004	07/02/2004	No
13	13	San Bernardino	42	F	WNF	06/09/2004	07/05/2004	No
14	14	San Bernardino	50	M	WNND	06/21/2004	07/05/2004	No
15	15	Riverside	43	F	WNND	06/22/2004	07/05/2004	No
16	16	Riverside	52	M	WNND	06/10/2004	07/06/2004	No
17	17	San Bernardino	15	M	WNND	06/30/2004	07/08/2004	No
18	18	San Bernardino	53	M	WNF	06/28/2004	07/09/2004	No
19	19	San Bernardino	22	M	WNND	06/28/2004	07/09/2004	No

Fig. 3.3. Using the `fix` function to edit the WNV surveillance data frame. Unfortunately, this approach does not facilitate documenting your edits. Source: California Department of Health Services, 2004

unless you assign it to the original or new object name. In contrast, changes made with the `fix` function are saved back to the original data object name. Therefore, be careful when you use the `fix` function because you may unintentionally overwrite data.

Now let's read in the WNV surveillance raw data as a data frame. Then, using the `fix` function, we will edit the first three records where the value for the syndrome variable is "Unk" and change it to NA for missing (Figure 3.3). We will also change "." to NA.

```
> wd <- read.table("http://www.medept.net/data/wnv/wnv2004raw.txt",
+ header = TRUE, sep = ",", as.is = TRUE)
> wd[wd$syndrome=="Unknown",][1:3,] #before edits (3 records)
  id county age sex syndrome date.onset date.tested death
128 128 Los Angeles 81 M Unknown 07/28/2004 08/11/2004 .
129 129 Riverside 44 F Unknown 07/25/2004 08/11/2004 .
133 133 Los Angeles 36 M Unknown 08/04/2004 08/11/2004 No
> fix(wd) #open R spreadsheet and make edits (see figure)
> wd[c(128, 129, 133),] #after edits (3 records)
  id county age sex syndrome date.onset date.tested death
128 128 Los Angeles 81 M NA 07/28/2004 08/11/2004 NA
129 129 Riverside 44 F NA 07/25/2004 08/11/2004 NA
133 133 Los Angeles 36 M NA 08/04/2004 08/11/2004 No
```

First, notice that in the `read.table` function `as.is=TRUE`. This means the data is read in without R making any changes to it. In other words, character vectors are not automatically converted to factors. We set the option because we knew we were going to edit and make corrections to the

data set, and create factors later. In this example, I manually started changing the missing values “Unknown” to NA (R’s representation of missing values). However, this manual approach would be very inefficient. A better approach is to specify which values in the data frame should be converted to NA. In the `read.table` function I should have set the option `na.string=c("Unknown", ".")`, converting the character strings “Unknown” and “.” into NA. Let’s replace the missing values with NAs upon reading the data file. `wd` `j`- `read.table("http://www.medepi.net/data/wnv2004raw.txt", header = TRUE, sep = ", ", as.is = TRUE, na.string=c("Unknown", "."))`

```
> wd <- read.table("http://www.medepi.net/data/wnv/wnv2004raw.txt",
+   Header = TRUE, sep = ", ", as.is = TRUE,
+   na.string=c("Unknown", "."))
> wd[c(128, 129, 133),] #verify change
   id   county age sex syndrome date.onset date.tested death
128 128 Los Angeles 81  M    <NA> 07/28/2004 08/11/2004 <NA>
129 129 Riverside 44  F    <NA> 07/25/2004 08/11/2004 <NA>
133 133 Los Angeles 36  M    <NA> 08/04/2004 08/11/2004 No
```

3.2.3 Vectorized approach

How do we make these and other changes after the data set has been read into R? Although using R’s spreadsheet function is convenient, we do not recommend it because manual editing is inefficient, your work cannot be easily audited and replicated, and documentation is poor. Instead use R’s vectorized approach. Let’s look at the distribution of responses for each variable to assess what needs to be “cleaned up,” in addition to converting missing values to NA.

```
> wd <- read.table("http://www.medepi.net/data/wnv/wnv2004raw.txt",
+   header = TRUE, sep = ", ", as.is = TRUE)
> str(wd)
'data.frame': 779 obs. of 8 variables:
 $ id      : int  1 2 3 4 5 6 7 8 9 10 ...
 $ county  : chr  "San Bernardino" "San Bernardino" ...
 $ age     : chr  "40" "64" "19" "12" ...
 $ sex     : chr  "F" "F" "M" "M" ...
 $ syndrome : chr  "WNF" "WNF" "WNF" "WNF" ...
 $ date.onset : chr  "05/19/2004" "05/22/2004" ...
 $ date.tested: chr  "06/02/2004" "06/16/2004" ...
 $ death   : chr  "No" "No" "No" "No" ...
> lapply(wd, table) #apply 'table' function to fields
$id
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
```

...
 768 769 770 771 772 773 774 775 776 777 778 779 780 781
 1 1 1 1 1 1 1 1 1 1 1 1 1 1

\$county

Butte	Fresno	Glenn	Imperial
7	11	3	1
Kern	Lake	Lassen	Los Angeles
59	1	1	306
Merced	Orange	Placer	Riverside
1	62	1	109
Sacramento	San Bernardino	San Diego	San Joaquin
3	187	2	2
Santa Clara	Shasta	Sn Luis Obispo	Tehama
1	5	1	10
Tulare	Ventura	Yolo	
3	2	1	

\$age

. 1 10 11 12 13 14 15 16 17 18 19 2 20 21 22 23 24 25 26
 6 1 1 1 3 2 3 3 1 4 6 5 1 4 2 3 6 8 3 9
 ...
 82 83 84 85 86 87 88 89 9 91 93 94
 10 5 6 4 2 2 1 6 1 4 1 1

\$sex

. F M
 2 294 483

\$syndrome

Unknown	WNF	WNND
105	391	283

\$date.onset

02/02/2005 05/14/2004 05/16/2004 05/19/2004 05/22/2004
 1 1 1 1 2
 ...
 10/28/2004 10/29/2004 10/30/2004 11/08/2004 11/12/2004
 2 1 1 4 2

```

$date.tested

01/21/2005 02/04/2005 02/23/2005 06/02/2004 06/16/2004
              1          1          1          1          4
...
11/29/2004 12/02/2004 12/03/2004 12/07/2004
              8          1          2          1

$death

.   No Yes
66 686 27

```

Here is what we learn. First, there are 779 observations and 781 id's; therefore, 3 observations were removed from the original data set. Second, we see that the variables age, sex, syndrome, and death have missing values that need to be converted to NAs. This can be done one field at a time, or for the whole data frame in one step. Here is the R code.

```

#individually
wd$age[wd$age=="."] <- NA
wd$sex[wd$sex=="."] <- NA
wd$syndrome[wd$syndrome=="Unknown"] <- NA
wd$death[wd$death=="."] <- NA

#or globally
wd[wd=="." | wd=="Unknown"] <- NA

```

After running the above code, let's evaluate one variable to verify the missing values were converted to NAs.

```

> table(wd$death)

No Yes
686 27
> table(wd$death, exclude=NULL)

No Yes <NA>
686 27 66

```

We also notice that the entry for one of the counties, San Luis Obispo, was misspelled (`Sn Luis Obispo`). We can use replacement to make the corrections:

```

> wd$County[wd$County=="Sn Luis Obispo"] <- "San Luis Obispo"

```

3.2.4 Text processing

On occasion, you will need to process and manipulate character vectors using a vectorized approach. For example, suppose you need to convert a character vector of dates from “mm/dd/yy” to “yyyy-mm-dd” (the international standard for dates). We’ll start by using the `substr` function. This function extracts characters from a character vector based on position.

```
> bd <- c("07/17/96", "12/09/00", "11/07/97")
> mon <- substr(bd, start=1, stop=2); mon
[1] "07" "12" "11"
> day <- substr(bd, 4, 5); day
[1] "17" "09" "07"
> yr <- as.numeric(substr(bd, 7, 8)); yr
[1] 96 0 97
> yr2 <- ifelse(yr<=19, yr+2000, yr+1900); yr2
[1] 1996 2000 1997
> bdfin <- paste(yr2, "-", mon, "-", day, sep=""); bdfin
[1] "1996-07-17" "2000-12-09" "1997-11-07"
```

In this example, we needed to convert “00” to “2000”, and “96” and “97” to “1996” and “1997”, respectively. The trick here was to coerce the character vector into a numeric vector so that 1900 or 2000 could be added to it. Using the `ifelse` function, for values ≤ 19 (arbitrarily chosen), 2000 was added, otherwise 1900 was added. The `paste` function was used to paste back the components into a new vector with the standard date format.

The `substr` function can also be used to replace characters in a character vector.

```
> bd
[1] "07/17/96" "12/09/00" "11/07/97"
> substr(bd, 3, 3) <- "-"
> substr(bd, 6, 6) <- "-"
> bd
[1] "07-17-96" "12-09-00" "11-07-97"
```

3.3 Sorting data

The `sort` function sorts a vector as expected:

```
> x <- sample(1:10, 10); x
[1] 4 3 6 1 7 9 5 8 2 10
> sort(x)
[1] 1 2 3 4 5 6 7 8 9 10
> sort(x, decreasing = TRUE) #reverse sort
[1] 10 9 8 7 6 5 4 3 2 1
```

Table 3.2. R functions for processing text in character vectors

Function	Description	Try these examples
nchar	Returns the number of characters in each element of a character vector	<code>x <- c("a", "ab", "abc", "abcd")</code> <code>nchar(x)</code>
substr	Extract or replace substrings in a character vector	#extraction <code>mon <- substr(some.dates, 1, 2); mon</code> <code>day <- substr(some.dates, 4, 5); day</code> <code>yr <- substr(some.dates, 7, 8); yr</code> #replacement <code>mdy <- paste(mon, day, yr); mdy</code> <code>substr(mdy, 3, 3) <- '/'</code> <code>substr(mdy, 6, 6) <- '/'</code> <code>mdy</code>
paste	Concatenate vectors after converting to character	<code>rd <- paste(mon, "/", day, "/", yr, sep="")</code> <code>rd</code>
strsplit	Split the elements of a character vector into substrings	<code>some.dates <- c("10/02/70", "02/04/67")</code> <code>some.dates</code> <code>strsplit(some.dates, "/")</code>

```
> rev(sort(x))           #reverse sort
[1] 10 9 8 7 6 5 4 3 2 1
```

However, if you want to sort one vector based on the ordering of elements from another vector, use the `order` function. The `order` function generates an indexing/repositioning vector. Study the following example:

```
> x <- sample(1:20, 5); x
[1] 18 10 6 13 11
> sort(x)           #sorts as expected
[1] 6 10 11 13 18
> y <- sample(1:20, 5); y
[1] 11 10 13 3 9
> order(y)         #4th element to 1st position, 5th to 2nd, etc.
[1] 4 5 2 1 3
> x[order(y)]      #use order(y) to sort elements of x
[1] 13 11 10 18 6
```

Based on this we can see that `sort(x)` is just `x[order(x)]`.

Now let us see how to use the `order` function for data frames. First, we'll create a small data set.

```

> sex <- rep(c("Male", "Female"), c(4, 4))
> ethnicity <- rep(c("White", "African American", "Latino",
+                   "Asian"), 2)
> age <- sample(1:100, 8)
> dat <- data.frame(age, sex, ethnicity)
> dat <- dat[sample(1:8, 8),] #randomly order rows
> dat
  age  sex      ethnicity
5  57 Female         White
8  93 Female         Asian
1   7  Male         White
4  65  Male         Asian
6  38 Female African American
3  27  Male         Latino
2  66  Male African American
7  72 Female         Latino

```

Okay, now we will sort the data frame based on the ordering of one field, and then the ordering of two fields:

```

> dat[order(dat$age),] #sort based on 1 variable
  age  sex      ethnicity
1   7  Male         White
3  27  Male         Latino
6  38 Female African American
5  57 Female         White
4  65  Male         Asian
2  66  Male African American
7  72 Female         Latino
8  93 Female         Asian
> dat[order(dat$sex, dat$age),] #sort based on 2 variables
  age  sex      ethnicity
6  38 Female African American
5  57 Female         White
7  72 Female         Latino
8  93 Female         Asian
1   7  Male         White
3  27  Male         Latino
4  65  Male         Asian
2  66  Male African American

```

3.4 Indexing (subsetting) data

For this section, please load the well known Oswego foodborne illness dataset:

```

> odat <- read.table("http://www.medepi.net/data/oswego.txt",
+   header = TRUE, as.is = TRUE, sep = "")
> str(odat)
'data.frame':   75 obs. of  21 variables:
 $ id          : int  2 3 4 6 7 8 9 10 14 16 ...
 $ age         : int  52 65 59 63 70 40 15 33 10 32 ...
 $ sex         : chr  "F" "M" "F" "F" ...
 $ meal.time   : chr  "8:00 PM" "6:30 PM" "6:30 PM" ...
 $ ill         : chr  "Y" "Y" "Y" "Y" ...
 $ onset.date  : chr  "4/19" "4/19" "4/19" "4/18" ...
 $ onset.time  : chr  "12:30 AM" "12:30 AM" ...
 $ baked.ham   : chr  "Y" "Y" "Y" "Y" ...
 ...
 $ vanilla.ice.cream : chr  "Y" "Y" "Y" "Y" ...
 $ chocolate.ice.cream: chr  "N" "Y" "Y" "N" ...
 $ fruit.salad   : chr  "N" "N" "N" "N" ...

```

3.4.1 Indexing

Now, we will practice indexing rows from this data frame. First, let's create a new data set that contains only cases. To index the rows with cases we need to generate a logical vector that is TRUE for every value of `odat$ill` that "is equivalent to" "Y". For "is equivalent to" we use the `==` relational operator.

```

> cases <- odat$ill=="Y"
> cases
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 ...
[73] FALSE FALSE FALSE
> odat.ca <- odat[cases, ]
> odat.ca[, 1:8]
   id age sex meal.time ill onset.date onset.time baked.ham
1   2  52  F   8:00 PM   Y      4/19   12:30 AM         Y
2   3  65  M   6:30 PM   Y      4/19   12:30 AM         Y
3   4  59  F   6:30 PM   Y      4/19   12:30 AM         Y
4   6  63  F   7:30 PM   Y      4/18   10:30 PM         Y
 ...
43  71  60  M   7:30 PM   Y      4/19    1:00 AM         N
44  72  18  F   7:30 PM   Y      4/19   12:00 AM         Y
45  74  52  M      <NA>   Y      4/19    2:15 AM         Y
46  75  45  F      <NA>   Y      4/18   11:00 PM         Y

```

It is very important to understand what we just did: we extracted the rows with cases by indexing the data frame with a logical vector.

Now, we combine relational operators with logical operators to extract rows based on multiple criteria. Let's create a data set with female cases, age less than the median age, and consumed vanilla ice cream.

```
> fem.cases.vic <- odat$ill=="Y" & odat$sex=="F" &
+   odat$vanilla.ice.cream=="Y" & odat$age < median(odat$age)
> odat.fcv <- odat[fem.cases.vic, ]
> odat.fcv[ , c(1:6, 19)]
```

	id	age	sex	meal.time	ill	onset.date	vanilla.ice.cream
8	10	33	F	7:00 PM	Y	4/18	Y
10	16	32	F	<NA>	Y	4/19	Y
13	20	33	F	<NA>	Y	4/18	Y
14	21	13	F	10:00 PM	Y	4/19	Y
18	27	15	F	10:00 PM	Y	4/19	Y
23	36	35	F	<NA>	Y	4/18	Y
31	48	20	F	7:00 PM	Y	4/19	Y
37	58	12	F	10:00 PM	Y	4/19	Y
40	65	17	F	10:00 PM	Y	4/19	Y
41	66	8	F	<NA>	Y	4/19	Y
42	70	21	F	<NA>	Y	4/19	Y
44	72	18	F	7:30 PM	Y	4/19	Y

In summary, we see that indexing rows of a data frame consists of using relational operators (<, >, <=, >=, ==, !=) and logical operators (&, |, !) to generate a logical vector for indexing the appropriate rows.

3.4.2 Subsetting

Subsetting a data frame using the `subset` function is equivalent to using logical vectors to index the data frame. In general, we prefer indexing because it is generalizable to indexing any R data object. However, the `subset` function is a convenient alternative for data frames. Again, let's create data set with female cases, age < median, and ate vanilla ice cream.

```
> odat.fcv <- subset(odat, subset = {ill=="Y" & sex=="F" &
+   vanilla.ice.cream=="Y" & age < median(odat$age)},
+   select = c(id:onset.date, vanilla.ice.cream))
> odat.fcv
```

	id	age	sex	meal.time	ill	onset.date	vanilla.ice.cream
8	10	33	F	7:00 PM	Y	4/18	Y
10	16	32	F	.	Y	4/19	Y
13	20	33	F	.	Y	4/18	Y
14	21	13	F	10:00 PM	Y	4/19	Y
18	27	15	F	10:00 PM	Y	4/19	Y
23	36	35	F	.	Y	4/18	Y
31	48	20	F	7:00 PM	Y	4/19	Y

```

37 58 12 F 10:00 PM Y 4/19 Y
40 65 17 F 10:00 PM Y 4/19 Y
41 66 8 F . Y 4/19 Y
42 70 21 F . Y 4/19 Y
44 72 18 F 7:30 PM Y 4/19 Y

```

In the `subset` function, the first argument is the data frame object name, the second argument (also called `subset`) evaluates to a logical vector, and third argument (called `select`) specifies the fields to keep. In the second argument,

```
subset = {...}
```

the curly brackets are included for convenience to group the logical and relational operations.

3.5 Transforming data

Transforming fields in a data frame is very common. The most common transformations include the following:

- Numerical transformation of a numeric vector
- Discretizing a numeric vector into categories or levels (“categorical variable”)
- Re-coding integers that represent levels of a categorical variable

For each of these, one must decide whether the newly created vector should be a new field in the data frame, overwrite the original field in the data frame, or not be a field in the data frame (but rather a vector object in the workspace). For the examples that follow load the well known Oswego foodborne illness dataset:

```
> odat <- read.table("http://www.medepi.net/data/oswego.txt",
+   header = TRUE, as.is = TRUE, sep = "")
```

3.5.1 Numerical transformation

```

> # transform age variable centering it
> # create new field in same data frame
> odat$age
[1] 52 65 59 63 70 40 15 33 10 32 62 36 33 13 7 3 59 15
...
[73] 17 36 14
> odat$age.centered <- odat$age - mean(odat$age)
> odat$age.centered
[1] 15.1866667 28.1866667 22.1866667 26.1866667
...

```

```
[73] -19.8133333 -0.8133333 -22.8133333
>
> # overwrite original field in same data frame (not recommended!!!)
> # odat$age <- odat$age - mean(odat$age)
>
> # create new vector in workspace; data frame remains unchanged
> age.centered <- odat$age - mean(odat$age)
> age.centered
 [1] 15.1866667 28.1866667 22.1866667 26.1866667
...
[73] -19.8133333 -0.8133333 -22.8133333
```

For convenience, the `transform` function facilitates the transformation of numeric vectors in a data frame. The `transform` function comes in handy when we plan on transforming many fields: we do not need to specify the data frame each time you refer to a field name. For example, the following lines are equivalent. Both add a new transformed field to the data frame.

```
odat$age.centered <- odat$age - mean(odat$age)
odat <- transform(odat, age.centered = age - mean(age))
```

3.5.2 Creating categorical variables (factors)

Now, reload the Oswego data set to recover the original `odat$age` field. We are going to create a new field with the following seven age categories (in years): < 1, 1 to 4, 5 to 14, 15 to 24, 25 to 44, 45 to 64, and 65+. We will demonstrate this using several methods:

Using `cut` function (preferred method)

```
> agecat <- cut(odat$age, breaks = c(0, 1, 5, 15, 25, 45,
+   65, 100))
> agecat
 [1] (45,65] (45,65] (45,65] (45,65] (65,100] (25,45]
...
[73] (15,25] (25,45] (5,15]
7 Levels: (0,1] (1,5] (5,15] (15,25] (25,45] ... (65,100]
```

Note that the `cut` function generated a factor with 7 levels for each interval. The notation `(15, 25]` means that the interval is open on the left boundary (> 15) and closed on the right boundary (≤ 25). However, for age categories, it makes more sense to have age boundaries closed on the left and open on the right: `[a, b)`. To change this we set the option `right = FALSE`

```
> agecat <- cut(odat$age, breaks = c(0, 1, 5, 15, 25, 45,
+   65, 100), right = FALSE)
```

```

> agecat
[1] [45,65) [65,100) [45,65) [45,65) [65,100) [25,45)
...
[73] [15,25) [25,45) [5,15)
7 Levels: [0,1) [1,5) [5,15) [15,25) [25,45) ... [65,100)
> table(agecat)
agecat
  [0,1) [1,5) [5,15) [15,25) [25,45) [45,65) [65,100)
      0      1      14      13      18      20      9

```

Okay, this looks good, but we can add labels since our readers may not be familiar with open and closed interval notation $[a, b)$.

```

> agelabs <- c("<1", "1 to 4", "5 to 14", "15 to 24",
+ "25 to 44", "45 to 64", "65+")
> agecat <- cut(odat$age, breaks = c(0, 1, 5, 15, 25, 45,
+ 65, 100), right = FALSE, labels = agelabs)
> agecat
[1] 45 to 64 65+      45 to 64 45 to 64 65+      25 to 44
...
[73] 15 to 24 25 to 44 5 to 14
7 Levels: <1 1 to 4 5 to 14 15 to 24 25 to 44 ... 65+
> table(agecat, case = odat$ill)
      case
agecat  N  Y
  <1      0  0
  1 to 4   0  1
  5 to 14   8  6
  15 to 24  5  8
  25 to 44  8 10
  45 to 64  5 15
  65+      3  6

```

Using indexing and assignment (replacement)

The `cut` function is the preferred method to create a categorical variable. However, suppose one does not know about the `cut` function. Applying basic R concepts always works!

```

> agegroup <- odat$age
> agegroup[odat$age<1] <- 1
> agegroup[odat$age>=1 & odat$age<5] <- 2
> agegroup[odat$age>=5 & odat$age<15] <- 3
> agegroup[odat$age>=15 & odat$age<25] <- 4
> agegroup[odat$age>=25 & odat$age<45] <- 5
> agegroup[odat$age>=45 & odat$age<65] <- 6

```

```

> agegroup[odat$age>=65] <- 7
> #create factor
> agelabs <- c("<1", "1 to 4", "5 to 14", "15 to 24",
+           "25 to 44", "45 to 64", "65+")
> agegroup <- factor(agegroup, levels = 1:7, labels = agelabs)
> agegroup
 [1] 45 to 64 65+      45 to 64 45 to 64 65+      25 to 44
...
[73] 15 to 24 25 to 44 5 to 14
7 Levels: <1 1 to 4 5 to 14 15 to 24 25 to 44 ... 65+
> table(case = odat$ill, agegroup)
  agegroup
case <1 1 to 4 5 to 14 15 to 24 25 to 44 45 to 64 65+
  N    0     0     8     5     8     5     3
  Y    0     1     6     8    10    15     6

```

In these previous examples, notice that `agegroup` is a factor object that is not a field in the `odat` data frame.

3.5.3 “Re-coding” levels of a categorical variable

In the previous example the categorical variable was a numeric vector (1, 2, 3, 4, 5, 6, 7) that was converted to a factor and provided labels (“<1”, “1 to 4”, “5 to 14”, ...). In fact, categorical variables are often represented by integers (for example, 0 = no, 1 = yes; or 0 = non-case, 1 = case) and provided labels. Often, ASCII text data files are integer codes that require a data dictionary to convert these integers into categorical variables in a statistical package. In R, keeping track of integer codes for categorical variables is unnecessary. Therefore, re-coding the underlying integer codes is also unnecessary; however, if you feel the need to do so, here’s how.

```

> # Create categorical variable
> ethlabs <- c("White", "Black", "Latino", "Asian")
> ethnicity <- sample(ethlabs, 100, replace = T)
> ethnicity <- factor(ethnicity, levels = ethlabs)
> ethnicity
 [1] Black Asian Latino White Black Asian White Black
...
[97] Black Black Asian Latino
Levels: White Black Latino Asian

```

The `levels` option allowed us to determine the display order, and the first level becomes the reference level in statistical models. To display the underlying numeric code use `unclass` function which preserves the levels attribute.⁴

⁴ The `as.integer` function also works but does not preserve the levels attribute.

```

> x <- unclass(ethnicity)
> x
 [1] 2 4 3 1 2 4 1 2 1 3 4 3 2 1 3 2 1 1 1 3 1 2 2 1 3 4 2 3
...
 [85] 2 2 3 3 3 3 1 3 4 4 1 1 2 2 4 3
attr(,"levels")
 [1] "White" "Black" "Latino" "Asian"

```

To recover the original factor,

```

> factor(x,labels=levels(x))
 [1] Black Asian Latino White Black Asian White
...
 [92] Latino Asian Asian White White Black Black
 [99] Asian Latino
Levels: White Black Latino Asian

```

Although one can extract the integer code, why would one need to do so? One is tempted to use the integer codes as a way to share data sets. However, we recommend not using the integer codes, but rather just provided the data in its native format⁵. This way, the raw data is more interpretable and eliminates the intermediate step of needing to label the integer code. Also, if the data dictionary is lost or not provided, the raw data is still interpretable.

In R, we can re-label the levels using the `levels` function and assigning to it a character vector of new labels. Make sure the order of the new labels corresponds to order of the factor levels.

```

> levels(ethnicity2) <- c("Caucasion", "African American",
+ "Hispanic", "Asian")
> table(ethnicity2)
ethnicity2
   Caucasion African American Hispanic Asian
          23             31          28    18

```

In R, we can re-order and re-label at the same time using the `levels` function and assigning to it a list.

```

> table(ethnicity)
ethnicity
 White Black Latino Asian
    23   31   28   18
> ethnicity3 <- ethnicity
> levels(ethnicity3) <- list(Hispanic = "Latino", Asian = "Asian",
+ Caucasion = "White", "African American" = "Black")
> table(ethnicity3)
ethnicity3

```

⁵ For example, <http://www.medepi.net/data/oswego.txt>

Hispanic	Asian	Caucasian	African American
28	18	23	31

The `list` function is necessary to assure the re-ordering. To re-order without re-labeling just do the following:

```
> table(ethnicity)
ethnicity
White Black Latino Asian
  23   31   28   18
> ethnicity4 <- ethnicity
> levels(ethnicity4) <- list(Latino = "Latino", Asian = "Asian",
+   White = "White", Black = "Black")
> table(ethnicity4)
ethnicity4
Latino Asian White Black
  28   18   23   31
```

In R, we can sort the factor levels by using the `factor` function in one of two ways:

```
> table(ethnicity)
ethnicity
White Black Latino Asian
  23   31   28   18
> ethnicity5a <- factor(ethnicity, sort(levels(ethnicity)))
> table(ethnicity5a)
ethnicity5a
Asian Black Latino White
  18   31   28   23
> ethnicity5b <- factor(as.character(ethnicity))
> table(ethnicity5b)
ethnicity5b
Asian Black Latino White
  18   31   28   23
```

In the first example, we assigned to the `levels` argument the sorted level names. In the second example, we started from scratch by coercing the original factor into a character vector which is then ordered alphabetically by default.

Setting factor reference level

The first level of a factor is the reference level for some statistical models (e.g., logistic regression). To set a different reference level use the `relevel` function.

```
> levels(ethnicity)
[1] "White" "Black" "Latino" "Asian"
> ethnicity6 <- relevel(ethnicity, ref = "Asian")
```

Table 3.3. Categorical variable represented as a factor or a set of dummy variables

Factor	Dummy variables		
Ethnicity	Asian	Black	Latino
White	0	0	0
Asian	1	0	0
Black	0	1	0
Latino	0	0	1

```
> levels(ethnicity6)
[1] "Asian" "White" "Black" "Latino"
```

As we can see, there is tremendous flexibility in dealing with factors without the need to “re-code” categorical variables. This approach facilitates reviewing your work and minimizes errors.

3.5.4 Use factors instead of dummy variables

A nonordered factor (nominal categorical variable) with k levels can also be represented with $k - 1$ dummy variables. For example, the `ethnicity` factor has four levels: white, Asian, black, and Latino. Ethnicity can also be represented using 3 dichotomous variables, each coded 0 or 1. For example, using white as the reference group, the dummy variables would be `asian`, `black`, and `latino` (see Table 3.3). The values of those three dummy variables (0 or 1) are sufficient to represent one of four possible ethnic categories. Dummy variables can be used in statistical models. However, in R, it is unnecessary to create dummy variables, just create a factor with the desired number of levels and set the reference level.

3.5.5 Conditionally transforming the elements of a vector

You can conditionally transform the elements of a vector using the `ifelse` function. This function works as follows: `ifelse(test, if test = TRUE do this, else do this)`.

```
> x <- sample(c("M", "F"), 10, replace = T); x
[1] "M" "F" "M" "F" "M" "F" "M" "F" "M" "F"
> y <- ifelse(x=="M", "Male", "Female")
> y
[1] "Male" "Female" "Male" "Female" "Male" "Female"
[7] "Male" "Female" "Male" "Female"
```

Table 3.4. R functions for transforming variables in data frames

Function	Description	Try these examples
<-	Transforming a vector and assigning it to a new data frame variable name	<pre>dat <- data.frame(id=1:3, x=c(0.5,1,2)); dat dat\$logx <- log(x) #creates new field dat</pre>
transform	Transform one or more variables from a data frame	<pre>dat <- data.frame(id=1:3, x=c(0.5,1,2)); dat dat <- transform(dat, logx = log(x)) dat</pre>
cut	Creates a factor by dividing the range of a numeric vector into intervals	<pre>age <- sample(1:100, 500, replace = TRUE) # cut into 2 intervals agecut <- cut(age, 2, right = FALSE) table(agecut) #cut using specified intervals agecut2 <- cut(age, c(0, 50 100), right = FALSE, include.lowest = TRUE) table(agecut2)</pre>
levels	Gives access to the levels attribute of a factor	<pre>sex <- sample(c("M","F","T"),500, replace=T) sex <- factor(sex) table(sex) # relabel each level; use same order levels(sex) <- c("Female", "Male", "Transgender") table(sex) # relabel/recombine levels(sex) <- c("Female", "Male", "Male") table(sex) # reorder and/or relabel levels(sex) <- list ("Men" = "Male", "Women" = "Female") table(sex)</pre>
relevel	Set the reference level for a factor	<pre>sex2 <- relevel(sex, ref = "Women") table(sex2)</pre>
ifelse	Conditionally operate on elements of a vector based on a test	<pre>age <- sample(1:100, 1000, replace = TRUE) agecat <- ifelse(age<=50, "<=50", ">50") table(agecat)</pre>

3.6 Merging data

In general, R's strength is not data management but rather data analysis. Because R can access and operate on multiple objects in the workspace it is generally not necessary to merge data objects into one data object in order to conduct analyses. On occasion, it may be necessary to merge two data frames into one data frames.

Data frames that contain data on individual subjects are generally of two types: (1) each row contains data collected on one and only one individual, or (2) multiple rows contain repeated measurements on individuals. The latter approach is more efficient at storing data. For example, here are two approaches to collecting multiple telephone numbers for two individuals.

```
> tab1
      name  wphone  fphone  mphone
1  Tomas Aragon 643-4935 643-2926 847-9139
2 Wayne Enanoria 643-4934   <NA>   <NA>
>
> tab2
      name telephone teletype
1  Tomas Aragon 643-4935   Work
2  Tomas Aragon 643-2926   Fax
3  Tomas Aragon 847-9139  Mobile
4 Wayne Enanoria 643-4934   Work
```

The first approach is represented by `tab1`, and the second approach by `tab2`⁶. Data is more efficiently stored in `tab2`, and adding new types of telephone numbers only requires assigning a new value (e.g., `Pager`) to the `teletype` field.

```
> tab2
      name telephone teletype
1  Tomas Aragon 643-4935   Work
2  Tomas Aragon 643-2926   Fax
3  Tomas Aragon 847-9139  Mobile
4 Wayne Enanoria 643-4934   Work
5  Tomas Aragon 719-1234  Pager
```

In both these data frames, an indexing field identifies an unique individual that is associated with each row. In this case, the `name` column is the indexing field for both data frames.

Now, let's look at an example of two related data frames that are linked by an indexing field. The first data frame contains telephone numbers for 5 employees and `fname` is the indexing field. The second data frame contains email addresses for 3 employees and `name` is the indexing field.

⁶ This approach is the basis for designing and implementing relational databases. A relational database consists of multiple tables linked by an indexing field.

```

> phone
  fname phonenum phonetype
1  Tomas 643-4935      work
2  Tomas 847-9139     mobile
3  Tomas 643-4926      fax
4  Chris 643-3932      work
5  Chris 643-4926      fax
6  Wayne 643-4934      work
7  Wayne 643-4926      fax
8   Ray  643-4933      work
9   Ray  643-4926      fax
10 Diana 643-3931      work
> email
  name          mail mailtype
1  Tomas  aragon@berkeley.edu    Work
2  Tomas  aragon@medepi.net      Personal
3  Wayne  enanoria@berkeley.edu   Work
4  Wayne  enanoria@idready.org    Work
5  Chris  csiador@berkeley.edu    Work
6  Chris  csiador@yahoo.com      Personal

```

To merge these two data frames use the merge function.

```

> dat <- merge(email, phone, by.x="name", by.y="fname")
> dat
  name          mail mailtype phonenum phonetype
1  Chris  csiador@berkeley.edu    Work 643-3932      work
2  Chris  csiador@yahoo.com      Personal 643-3932      work
3  Chris  csiador@berkeley.edu    Work 643-4926      fax
4  Chris  csiador@yahoo.com      Personal 643-4926      fax
5  Tomas  aragon@berkeley.edu    Work 643-4935      work
6  Tomas  aragon@medepi.net      Personal 643-4935      work
7  Tomas  aragon@berkeley.edu    Work 847-9139     mobile
8  Tomas  aragon@medepi.net      Personal 847-9139     mobile
9  Tomas  aragon@berkeley.edu    Work 643-4926      fax
10 Tomas  aragon@medepi.net      Personal 643-4926      fax
11 Wayne  enanoria@berkeley.edu   Work 643-4934      work
12 Wayne  enanoria@idready.org    Work 643-4934      work
13 Wayne  enanoria@berkeley.edu   Work 643-4926      fax
14 Wayne  enanoria@idready.org    Work 643-4926      fax
> dat <- merge(phone, email, by.x="fname", by.y="name")
> dat
  fname phonenum phonetype          mail mailtype
1  Chris 643-3932      work  csiador@berkeley.edu    Work
2  Chris 643-4926      fax  csiador@berkeley.edu    Work
3  Chris 643-3932      work  cvsiador@yahoo.com      Personal

```

4	Chris	643-4926	fax	cvsiador@yahoo.com	Personal
5	Tomas	643-4935	work	aragon@berkeley.edu	Work
6	Tomas	847-9139	mobile	aragon@berkeley.edu	Work
7	Tomas	643-4926	fax	aragon@berkeley.edu	Work
8	Tomas	643-4935	work	aragon@medepi.net	Personal
9	Tomas	847-9139	mobile	aragon@medepi.net	Personal
10	Tomas	643-4926	fax	aragon@medepi.net	Personal
11	Wayne	643-4934	work	enanoria@berkeley.edu	Work
12	Wayne	643-4926	fax	enanoria@berkeley.edu	Work
13	Wayne	643-4934	work	enanoria@idready.org	Work
14	Wayne	643-4926	fax	enanoria@idready.org	Work

The `by.x` and `by.y` options identify the indexing fields. By default, R selects the rows from the two data frames that is based on the *intersection* of the indexing fields (`by.x`, `by.y`). To merge the *union* of the indexing fields, set `all=TRUE`:

```
> dat <- merge(phone, email, by.x="fname", by.y="name",
+             all=TRUE)
> dat
```

	fname	phonenum	phonetype	mail	mailtype
1	Chris	643-3932	work	csiador@berkeley.edu	Work
2	Chris	643-4926	fax	csiador@berkeley.edu	Work
3	Chris	643-3932	work	csiador@yahoo.com	Personal
4	Chris	643-4926	fax	csiador@yahoo.com	Personal
5	Diana	643-3931	work	<NA>	<NA>
6	Ray	643-4933	work	<NA>	<NA>
7	Ray	643-4926	fax	<NA>	<NA>
8	Tomas	643-4935	work	aragon@berkeley.edu	Work
9	Tomas	847-9139	mobile	aragon@berkeley.edu	Work
10	Tomas	643-4926	fax	aragon@berkeley.edu	Work
11	Tomas	643-4935	work	aragon@medepi.net	Personal
12	Tomas	847-9139	mobile	aragon@medepi.net	Personal
13	Tomas	643-4926	fax	aragon@medepi.net	Personal
14	Wayne	643-4934	work	enanoria@berkeley.edu	Work
15	Wayne	643-4926	fax	enanoria@berkeley.edu	Work
16	Wayne	643-4934	work	enanoria@idready.org	Work
17	Wayne	643-4926	fax	enanoria@idready.org	Work

To “reshape” tabular data look up and study the `reshape` and `stack` functions.

3.7 Directing output to an external file

3.8 Working with missing values

```

NA, NAN
na.fail(object, ...)
na.omit(object, ...)
na.exclude(object, ...)
na.pass(object, ...)
na.action
na.omit
is.na
is.nan

```

3.9 Working with dates and times

There are 60 seconds in 1 minute, 60 minutes in 1 hour, 24 hours in 1 day, 7 days in 1 week, and 365 days in 1 year (except every 4th year we have a leap year with 366 days). Although this seems straightforward, doing numerical calculations with these time measures is not. Fortunately, computers make this much easier. Functions to deal with dates are available in the **base**, **chron**, and **survival** packages.

Summarized in Figure 3.4 on the following page is the relationship between recorded data (calendar dates and times) and their representation in R as date-time class objects (**Date**, **POSIXlt**, **POSIXct**). The **as.Date** function converts a calendar date into a **Date** class object. The **strptime** function converts a calendar date and time into a date-time class object (**POSIXlt**, **POSIXct**). The **as.POSIXlt** and **as.POSIXct** functions convert date-time class objects into **POSIXlt** and **POSIXct**, respectively.

The **format** function converts date-time objects into human legible character data such as dates, days, weeks, months, times, etc. These functions are discussed in more detail in the paragraphs that follow.

3.9.1 Date functions in the base package

as.Date

Let's start with simple date calculations. The **as.Date** function in R converts calendar dates (e.g., 11/2/1949) into a **Date** objects—a numeric vector of class **Date**. The numeric information is the number of days since January 1, 1970—also called Julian dates. However, because calendar date data can come in a variety of formats, we need to specify the format so that **as.Date** does the correct conversion. Study the following analysis carefully.

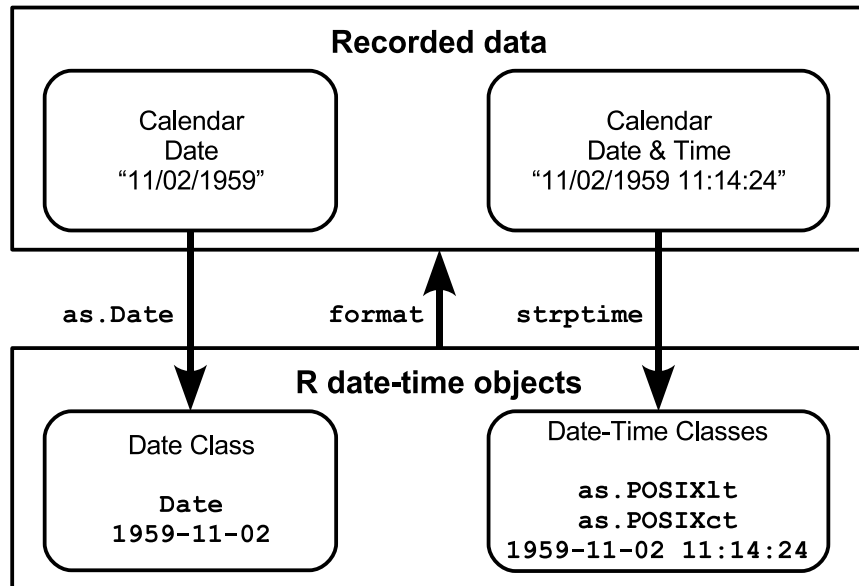


Fig. 3.4. Displayed are functions to convert calendar date and time data into R date-time classes (`as.Date`, `strptime`, `as.POSIXlt`, `as.POSIXct`), and the `format` function converts date-time objects into character dates, days, weeks, months, times, etc.

```
> bdays <- c("11/2/1959", "1/1/1970")
> bdays
[1] "11/2/1959" "1/1/1970"
> #convert to Julian dates
> bdays.julian <- as.Date(bdays, format = "%m/%d/%Y")
> bdays.julian
[1] "1959-11-02" "1970-01-01"
> #display Julian dates
> as.numeric(bdays.julian)
[1] -3713    0
> #calculate age as of today's date
> date.today <- Sys.Date()
> date.today
[1] "2005-09-25"
> age <- (date.today - bdays.julian)/365.25
> age
Time differences of 45.89733, 35.73169 days
> #the display of 'days' is not correct
> #truncate number to get "age"
> age2 <- trunc(as.numeric(age))
> age2
```

```
[1] 45 35
> #create date frame
> bd <- data.frame(Birthday = bdays, Standard = bdays.julian,
+   Julian = as.numeric(bdays.julian), Age = age2)
> bd
  Birthday Standard Julian Age
1 11/2/1959 1959-11-02 -3713 45
2  1/1/1970 1970-01-01     0 35
```

To summarize, `as.Date` converted the character vector of calendar dates into Julian dates (days since 1970-01-01) displayed in a standard format (yyyy-mm-dd). The Julian dates can be used in numerical calculations. To see the Julian dates use `as.numeric` or `julian` function. Because the calendar dates to be converted can come in a diversity of formats (e.g., November 2, 1959; 11-02-59; 11-02-1959; 02Nov59), one must specify the `format` option in `as.Date`. Below are selected format options; for a complete list see `help(strptime)`.

```
"%a" Abbreviated weekday name.
"%A" Full weekday name.
"%b" Abbreviated month name.
"%B" Full month name.
"%d" Day of the month as decimal number (01-31)
"%j" Day of year as decimal number (001-366).
"%m" Month as decimal number (01-12).
"%U" Week of the year as decimal number (00-53) using the
      first Sunday as day 1 of week 1.
"%w" Weekday as decimal number (0-6, Sunday is 0).
"%W" Week of the year as decimal number (00-53) using the
      first Monday as day 1 of week 1.
"%y" Year without century (00-99). If you use this on input,
      which century you get is system-specific. So don't!
      Often values up to 69 (or 68) are prefixed by 20 and
      70-99 by 19.
"%Y" Year with century.
```

Here are some examples of converting dates with different formats:

```
> as.Date("November 2, 1959", format = "%B %d, %Y")
[1] "1959-11-02"
> as.Date("11/2/1959", format = "%m/%d/%Y")
[1] "1959-11-02"
> #caution using 2-digit year
> as.Date("11/2/59", format = "%m/%d/%y")
[1] "2059-11-02"
> as.Date("02Nov1959", format = "%d%b%Y")
[1] "1959-11-02"
> #caution using 2-digit year
```

```

> as.Date("02Nov59", format = "%d%b%y")
[1] "2059-11-02"
> #standard format does not require format option
> as.Date("1959-11-02")
[1] "1959-11-02"

```

Notice how Julian dates can be used like any integer:

```

> as.Date("2004-01-15"):as.Date("2004-01-23")
[1] 12432 12433 12434 12435 12436 12437 12438 12439 12440
> seq(as.Date("2004-01-15"), as.Date("2004-01-18"), by = 1)
[1] "2004-01-15" "2004-01-16" "2004-01-17" "2004-01-18"

```

weekdays, months, quarters, julian

Use the `weekdays`, `months`, `quarters`, or `julian` functions to extract information from `Date` and other date-time objects in R.

```

> mydates <- c("2004-01-15", "2004-04-15", "2004-10-15")
> mydates <- as.Date(mydates)
> weekdays(mydates)
[1] "Thursday" "Thursday" "Friday"
> months(mydates)
[1] "January" "April" "October"
> quarters(mydates)
[1] "Q1" "Q2" "Q4"
> julian(mydates)
[1] 12432 12523 12706
attr("origin")
[1] "1970-01-01"

```

strptime

So far we have worked with calendar dates; however, we also need to be able to work with times of the day. Whereas `as.Date` only works with calendar dates, the `strptime` function will accept data in the form of calendar dates and times of day (HH:MM:SS, where H = hour, M = minutes, S = seconds). For example, let's look at the Oswego foodborne ill outbreak that occurred in 1940. The source of the outbreak was attributed to the church supper that was served on April 18, 1940. The food was available for consumption from 6 pm to 11 pm. The onset of symptoms occurred on April 18th and 19th. The meal consumption times and the illness onset times were recorded.

```

> odat <- read.table("http://www.medepi.net/data/oswego.txt",
+   sep = "", header = TRUE, as.is = TRUE)
> str(odat)
'data.frame': 75 obs. of 21 variables:

```

```

$ id      : int  2 3 4 6 7 8 9 10 14 16 ...
$ age     : int  52 65 59 63 70 40 15 33 10 32 ...
$ sex     : chr  "F" "M" "F" "F" ...
$ meal.time : chr  "8:00 PM" "6:30 PM" "7:30 PM" ...
$ ill     : chr  "Y" "Y" "Y" "Y" ...
$ onset.date : chr  "4/19" "4/19" "4/19" "4/18" ...
$ onset.time : chr  "12:30 AM" "10:30 PM" ...
...
$ vanilla.ice.cream : chr  "Y" "Y" "Y" "Y" ...
$ chocolate.ice.cream: chr  "N" "Y" "Y" "N" ...
$ fruit.salad      : chr  "N" "N" "N" "N" ...

```

To calculate the incubation period, for ill individuals, we need to subtract the meal consumption times (occurring on 4/18) from the illness onset times (occurring on 4/18 and 4/19). Therefore, we need two date-time objects to do this arithmetic. First, let's create a date-time object for the meal times:

```

> # look at existing data for meals
> odat$meal.time[1:5]
[1] "8:00 PM" "6:30 PM" "6:30 PM" "7:30 PM" "7:30 PM"
> # create character vector with meal date and time
> mdt <- paste("4/18/1940", odat$meal.time)
> mdt[1:4]
[1] "4/18/1940 8:00 PM" "4/18/1940 6:30 PM"
[3] "4/18/1940 6:30 PM" "4/18/1940 7:30 PM"
> # convert into standard date and time
> meal.dt <- strptime(mdt, format = "%m/%d/%Y %I:%M %p")
> meal.dt[1:4]
[1] "1940-04-18 20:00:00" "1940-04-18 18:30:00"
[3] "1940-04-18 18:30:00" "1940-04-18 19:30:00"
> # look at existing data for illness onset
> odat$onset.date[1:4]
[1] "4/19" "4/19" "4/19" "4/18"
> odat$onset.time[1:4]
[1] "12:30 AM" "12:30 AM" "12:30 AM" "10:30 PM"
> # create vector with onset date and time
> odt <- paste(paste(odat$onset.date, "/1940", sep=""),
+   odat$onset.time)
> odt[1:4]
[1] "4/19/1940 12:30 AM" "4/19/1940 12:30 AM"
[3] "4/19/1940 12:30 AM" "4/18/1940 10:30 PM"
> # convert into standard date and time
> onset.dt <- strptime(odt, "%m/%d/%Y %I:%M %p")
> onset.dt[1:4]
[1] "1940-04-19 00:30:00" "1940-04-19 00:30:00"
[3] "1940-04-19 00:30:00" "1940-04-18 22:30:00"

```

```

> # calculate incubation period
> incub.period <- onset.dt - meal.dt
> incub.period
Time differences of 4.5, 6.0, 6.0, 3.0, 3.0, 6.5, 3.0, 4.0,
6.5, NA, NA, NA, NA, 3.0, NA, NA, NA, 3.0, NA, NA,
3.0, 3.0, NA, NA, 3.0, NA, NA, NA, NA, NA, 6.0, NA,
4.0, NA, NA, NA, 3.0, 7.0, 4.0, 3.0, NA, NA, 5.5, 4.5,
NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,
NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,
NA, NA, NA, NA, NA, NA, NA, NA hours
> mean(incub.period, na.rm = T)
Time difference of 4.295455 hours
> median(incub.period, na.rm = T)
Error in Summary.difftime(..., na.rm = na.rm) :
  sum not defined for "difftime" objects
> # try 'as.numeric' on 'incub.period'
> median(as.numeric(incub.period), na.rm = T)
[1] 4

```

To summarize, we used `strptime` to convert the meal consumption date and times and illness onset dates and times into date-time objects (`meal.dt` and `onset.dt`) that can be used to calculate the incubation periods by simple subtraction (and assigned name `incub.period`).

Notice that `incub.period` is an atomic object of class `difftime`:

```

> str(incub.period)
Class 'difftime' atomic [1:75] 4.5 6 6 3 3 6.5 3 4 NA ...
..- attr(*, "tzone")= chr ""
..- attr(*, "units")= chr "hours"

```

This is why we had trouble calculating the median (which should not be the case). We got around this problem by coercion using `as.numeric`:

```

> as.numeric(incub.period)
[1] 4.5 6.0 6.0 3.0 3.0 6.5 3.0 4.0 6.5 NA NA NA NA 3.0
...
[71] NA NA NA NA NA

```

Now, what kind of objects were created by the `strptime` function?

```

> str(meal.dt)
'POSIXlt', format: chr [1:75] "1940-04-18 18:30:00" ...
> str(onset.dt)
'POSIXlt', format: chr [1:75] "1940-04-19 00:30:00" ...

```

The `strptime` function produces a named list of class `POSIXlt`. POSIX stands for “Portable Operating System Interface,” and “lt” stands for “legible time”.⁷

POSIXlt and POSIXct

The `POSIXlt` list contains the date-time data in human readable forms. The named list contains the following vectors:

```
'sec'    0-61: seconds
'min'    0-59: minutes
'hour'   0-23: hours
'mday'   1-31: day of the month
'mon'    0-11: months after the first of the year.
'year'   Years since 1900.
'wday'   0-6 day of the week, starting on Sunday.
'yday'   0-365: day of the year.
'isdst'  Daylight savings time flag. Positive if in force,
         zero if not, negative if unknown.
```

Let's examine the `onset.dt` object we created from the Oswego data.

```
> is.list(onset.dt)
[1] TRUE
> names(onset.dt)
[1] "sec" "min" "hour" "mday" "mon" "year" "wday"
[8] "yday" "isdst"
> onset.dt$min
[1] 30 30 30 30 30 0 0 0 0 30 30 15 0 0 0 45 45 0
...
[73] NA NA NA
> onset.dt$hour
[1] 0 0 0 22 22 2 1 23 2 10 0 22 22 1 23 21 21 1
...
[73] NA NA NA
> onset.dt$mday
[1] 19 19 19 18 18 19 19 18 19 19 19 18 18 19 18 18 18 19
...
[73] NA NA NA
> onset.dt$mon
[1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
...
[73] NA NA NA
> onset.dt$year
```

⁷ For more information visit the Portable Application Standards Committee site at <http://www.pasc.org/>

```

[1] 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
...
[73] NA NA NA
> onset.dt$yday
[1] 5 5 5 4 4 5 5 4 5 5 5 4 4 5 4 4 4 5
...
[73] NA NA NA
> onset.dt$yday
[1] 109 109 109 108 108 109 109 108 109 109 109 108 108 109
...
[71] NA NA NA NA NA

```

The POSIXlt list contains useful date-time information; however, it is not in a convenient form for storing in a data frame. Using `as.POSIXct` we can convert it to a “continuous time” object that contains the number of seconds since 1970-01-01 00:00:00. `as.POSIXlt` coerces a date-time object to POSIXlt.

```

> onset.dt.ct <- as.POSIXct(onset.dt)
> onset.dt.ct[1:5]
[1] "1940-04-19 00:30:00 Pacific Daylight Time"
[2] "1940-04-19 00:30:00 Pacific Daylight Time"
[3] "1940-04-19 00:30:00 Pacific Daylight Time"
[4] "1940-04-18 22:30:00 Pacific Daylight Time"
[5] "1940-04-18 22:30:00 Pacific Daylight Time"
> as.numeric(onset.dt.ct[1:5])
[1] -937326600 -937326600 -937326600 -937333800 -937333800

```

format

Whereas the `strptime` function converts a character vector of date-time information into a date-time object, the `format` function converts a date-time object into a character vector. The `format` function gives one great flexibility in converting date-time objects into numerous outputs (e.g., day of the week, week of the year, day of the year, month of the year, year). Selected date-time format options are listed on page 141, for a complete list see `help(strptime)`.

For example, in public health, reportable communicable diseases are often reported by “disease week” (this could be week of reporting or week of symptom onset). This information is easily extracted from R date-time objects. For weeks starting on Sunday use the “%U” option in the `format` function, and for weeks starting on Monday use the “%W” option.

```

> decjan <- seq(as.Date("2003-12-15"), as.Date("2004-01-15"),
+             by =1)
> decjan
[1] "2003-12-15" "2003-12-16" "2003-12-17" "2003-12-18"
...

```

Table 3.5. R functions for exporting data objects

Function	Description	Try these examples
write.table	Write tabular data as a	<code>data(infert)</code>
write.csv	data frame to an ASCII text file	<code>write.table(infert, "infert.dat")</code> <code>write.csv(infert, "infert.csv")</code>
dump	“Dumps” list of R objects as R code to an ASCII text file; read back in using <code>source</code> function	<code>data(Titanic)</code> <code>dump("Titanic", "titanic.R")</code>
dput & dget	Writes an R object as R code (but without the object name) to the console, or an ASCII text file; read file back in using <code>dget</code> function	<code>data(Titanic)</code> <code>dput(Titanic, "titanic.R")</code>
write	Write matrix elements to an ASCII text file	<code>x <- matrix(1:4, 2, 2)</code> <code>write(t(x), "x.txt")</code>
save	“Saves” list of R objects as binary <code>filename.Rdata</code> file; read back in using <code>load</code> function	<code>data(Titanic)</code> <code>save(Titanic, file="titanic.Rdata")</code>

```
[29] "2004-01-12" "2004-01-13" "2004-01-14" "2004-01-15"
> disease.week <- format(decjan, "%U")
> disease.week
 [1] "50" "50" "50" "50" "50" "50" "51" "51" "51" "51" "51"
 [12] "51" "51" "52" "52" "52" "52" "00" "00" "00" "01" "01"
 [23] "01" "01" "01" "01" "01" "02" "02" "02" "02" "02"
```

3.9.2 Date functions in the `chron` and `survival` packages

The `chron` and `survival` packages have customized functions for dealing with dates. Both packages come with the default R installation. To learn more about date and time classes read R News, Volume 4/1, June 2004⁸.

3.10 Exporting data objects

3.10.1 Exporting to an ASCII text file

The `write.table` function

The following code:

⁸ <http://cran.r-project.org/doc/Rnews>

```
data(infert)
write.table(infert, "/Users/tja/temp/infert.dat")
```

produces this ASCII text file:

```
"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
"5" "6-11yrs" 35 3 1 1 1 5 32
...
```

The following code:

```
write.csv(infert, "/Users/tja/temp/infert.csv")
```

produces this ASCII text file:

```
", "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
"5", "6-11yrs", 35, 3, 1, 1, 1, 5, 32
...
```

The dump function

```
udat2 <- array(c(8, 98, 5, 115, 22, 76, 16, 69), dim = c(2, 2, 2),
  dimnames = list(Status = c("Dead", "Survived"),
    Treatment = c("Tolbutamide", "Placebo"),
    Age.Group = c("<55", "55+")))
udat <- apply(udat2, 1:2, sum)
udat; udat2

dump("udat2", "/Users/tja/temp/udat2.R")
```

The dput function

```
> dput(udat2)
structure(c(8, 98, 5, 115, 22, 76, 16, 69), .Dim = c(2L, 2L,
2L), .Dimnames = structure(list(Status = c("Dead", "Survived"
), Treatment = c("Tolbutamide", "Placebo"), Age.Group = c("<55",
"55+")), .Names = c("Status", "Treatment", "Age.Group")))

dput(udat2, "/Users/tja/temp/udat2.txt")
```

The write function

```
write(t(udat), "/Users/tja/temp/udat.txt", ncol=2)
```

3.10.2 Exporting to a binary file**The save function**

```
save(Titanic, file="/Users/tja/temp/titanic.Rdata")
```

3.11 Working with regular expressions

3.12 Problems

- Using a text editor and the data from Table 3.1 on page 107 create the following data frame:

```
> dat
  Status Treatment Agegrp Freq
1   Dead Tolbutamide  <55    8
2 Survived Tolbutamide  <55   98
3   Dead   Placebo    <55    5
4 Survived   Placebo  <55  115
5   Dead Tolbutamide  55+   22
6 Survived Tolbutamide  55+   76
7   Dead   Placebo    55+   16
8 Survived   Placebo    55+   69
```

ANSWER

```
#answer 1
udat <- array(c(8, 98, 5, 115, 22, 76, 16, 69), dim =
  c(2, 2, 2),
  dimnames = list(Status = c("Dead", "Survived"),
    Treatment = c("Tolbutamide", "Placebo"),
    Agegrp = c("<55", "55+")))
dat <- data.frame(as.table(udat))
dat

#answer 2
Status <- rep(c("Dead", "Survived"), 4)
Treatment <- rep(rep(c("Tolbutamide", "Placebo"),
  c(2, 2)), 2)
Agegrp <- rep(c("<55", "55+"), c(4, 4))
Freq <- c(8, 98, 5, 115, 22, 76, 16, 69)
dat <- data.frame(Status, Treatment, Agegrp, Freq)
dat

#answer 2b, equivalent to 2a
dat <- data.frame(
  Status = rep(c("Dead", "Survived"), 4),
  Treatment = rep(rep(c("Tolbutamide", "Placebo"),
    c(2, 2)), 2),
  Agegrp = rep(c("<55", "55+"), c(4, 4)),
  Freq = c(8, 98, 5, 115, 22, 76, 16, 69)
)
dat
```

2. Select 3 to 5 classmates and collect data on first name, last name, affiliation, two email addresses, and today's date. Using your text editor, create a data frame with this data.

ANSWER: See chapter

3. Review the United States data on AIDS cases by year available at <http://www.medeppi.net/data/aids.txt>. Read this data into a data frame. Graph a calendar time series of AIDS cases.

```
# Hint
plot(x, y, type = "l", xlab = "x axis label", lwd = 2,
     ylab = "y axis label", main = "main title")
```

ANSWER:

```
adat <- read.table("http://www.medeppi.net/data/aids.txt", header=TRUE,
                  sep=" ", na.strings=".")
head(adat)
plot(adat$year, adat$cases, type = "l", xlab = "Year", lwd = 2,
     ylab = "Cases", main = "Reported AIDS Cases in United States, 1980--2003")
```

4. Review the United States data on measles cases by year available at <http://www.medeppi.net/data/measles.txt>. Read this data into a data frame. Graph a calendar time series of measles cases using an arithmetic and semi-logarithmic scale.

```
# Hint
plot(x, y, type = "l", lwd = 2, xlab = "x axis label",
     ylab="y axis label", main = "main title")
plot(x, y, type = "l", lwd = 2, xlab = "x axis label", log = "y",
     ylab="y axis label", main = "main title")
```

ANSWER:

```
mdat <- read.table("http://www.medeppi.net/data/measles.txt",
                  header=TRUE, sep=" ")
head(mdat)
plot(mdat$year, mdat$cases, type = "l", xlab = "Year", lwd = 2,
     ylab = "Cases",
     main = "Reported Measles Cases in United States, 1980--2003")
plot(mdat$year, mdat$cases, type = "l", xlab = "Year", lwd = 2,
     log = "y", ylab = "Cases",
     main = "Reported Measles Cases in United States, 1980--2003")
```

5. Review the United States data on hepatitis B cases by year available at <http://www.medeppi.net/data/hepb.txt>. Read this data into a data frame. Using the R code below, plot a times series of AIDS and hepatitis B cases.

```
matplot(hepb$year, cbind(hepb$cases, aids$cases),
       type="l", lwd=2, xlab="Year", ylab="Cases",
```

Table 3.6. Data dictionary for Evans data set

Variable	Variable name	Variable type	Possible values
id	Subject identifier	Integer	
chd	Coronary heart disease	Categorical-nominal	0 = no 1 = yes
cat	Catecholamine level	Categorical-nominal	0 = normal 1 = high
age	Age	Continuous	years
chl	Cholesterol	Continuous	> 0
smk	Smoking status	Categorical-nominal	0 = never smoked 1 = ever smoked
ecg	Electrocardiogram	Categorical-nominal	0 = no abnormality 1 = abnormality
dbp	Diastolic blood pressure	Continuous	mm Hg
sbp	Systolic blood pressure	Continuous	mm Hg
hpt	High blood pressure	Categorical-nominal	0 = no 1 = yes (dbp \geq 95 or sbp \geq 160)
ch	cat \times hpt	Categorical	product term
cc	cat \times chl	Continuous	product term

```

main="Reported cases of Hepatitis B and AIDS,
United States, 1980-2003")
legend(1980, 100000, legend=c("Hepatitis B", "AIDS"),
      lwd=2, lty=1:2, col=1:2)

```

6. Review data from the Evans cohort study in which 609 white males were followed for 7 years, with coronary heart disease as the outcome of interest (<http://www.medepi.net/data/evans.txt>). The data dictionary is provided in Table 3.6.

- a) Recode the binary variables (0, 1) into factors with 2 levels.
- b) Discretized age into a factor with more than 2 levels.
- c) Create a new hypertension categorical variable based on the current classification scheme⁹:
 - Normal: SBP < 120 and DBP < 80;
 - Prehypertension: SBP=[120, 140) or DBP=[80, 90);
 - Hypertension-Stage 1: SBP=[140, 160) or DBP=[90, 100); and
 - Hypertension-Stage 2: SBP \geq 160 or DBP \geq 100.
- d) Using R, construct a contingency table comparing the old and new hypertension variables.

ANSWER:

⁹ <http://www.nhlbi.nih.gov/guidelines/hypertension/phycard.pdf>

```

edat <- read.table("http://www.medepi.net/data/evans.txt",
  header = TRUE, sep="")
str(edat)
#
table(edat$chd)
edat$chd2 <- factor(edat$chd, levels = 0:1,
  labels = c("No", "Yes"))
table(edat$chd2)
#
table(edat$cat)
edat$cat2 <- factor(edat$cat, levels = 0:1,
  labels = c("Normal", "High"))
table(edat$cat2)
#
table(edat$smk)
edat$smk2 <- factor(edat$smk, levels = 0:1,
  labels = c("Never", "Ever"))
table(edat$smk2)
#
table(edat$ecg)
edat$ecg2 <- factor(edat$ecg, levels = 0:1,
  labels = c("Normal", "Abnormal"))
table(edat$ecg2)
#
table(edat$hpt)
edat$hpt2 <- factor(edat$hpt, levels = 0:1,
  labels = c("No", "Yes"))
table(edat$hpt2)

```

ANSWER:

```

quantile(edat$age)
edat$age4 <- cut(edat$age, quantile(edat$age),
  right = FALSE, include.lowest = TRUE)
table(edat$age4)

```

ANSWER:

```

hptnew <- rep(NA, nrow(edat))
normal <- edat$sbp<120 & edat$dbp<80
hptnew[normal] <- 1
prehyp <- (edat$sbp>=120 & edat$sbp<140) |
  (edat$dbp>=80 & edat$dbp<90)
hptnew[prehyp] <- 2
stage1 <- (edat$sbp>=140 & edat$sbp<160) |
  (edat$dbp>=90 & edat$dbp<100)
hptnew[stage1] <- 3

```

```

stage2 <- edat$sbp>=160 | edat$dbp>=100
hptnew[stage2] <- 4
edat$hpt4 <- factor(hptnew, levels=1:4,
  labels=c("Normal", "PreHTN", "HTN.Stage1", "HTN.Stage2"))
table(edat$hpt4)

```

ANSWER:

```
table("Old HTN"=edat$hpt2, "New HTN"=edat$hpt4)
```

7. Review the California 2004 surveillance data on human West Nile virus cases available at <http://www.medept.net/data/wnv/wnv2004raw.txt>. Read in the data, taking into account missing values. Convert the calendar dates into the international standard format. Using the `write.table` function export the data as an ASCII text file.

ANSWER

```

wdat <- read.table("http://www.medept.net/data/wnv/wnv2004raw.txt",
  header=TRUE, sep=",", as.is=TRUE, na.strings=c(".", "Unknown"))
str(wdat)
wdat$date.onset2 <- as.Date(wdat$date.onset, format="%m/%d/%Y")
wdat$date.tested2 <- as.Date(wdat$date.tested, format="%m/%d/%Y")
write.table(wdat, "c:/temp/wnvdat.txt", sep=",", row.names=FALSE)

```