An Introduction to the Fundamentals & Functionality of the R Programming Language

— Part II: The Nuts & Bolts —

Theresa A Scott, MS
Biostatistician III
Department of Biostatistics
Vanderbilt University

theresa.scott@vanderbilt.edu

Table of Contents

Preface

We will be using the PBC data set that we introduced in the first document and some functions from the `Hmisc` add-on package. Therefore, make sure you load the `Hmisc` package using the `library()` function, change to (ie, set) the correct working directory, read-in our `pbc` data set, and make changes to the `pbc` data frame using the `Hmisc` package's `upData()` function. The code to do so is given in `Scott.IntroToR.II.R` code file. The 'contents' of the resulting updated `pbc` data frame should be the same as the output of the `contents()` function invocation below.

```
> contents(pbc)

Data frame:pbc        100 observations and 14 variables    Maximum # NAs:32

                             Labels Units Levels Storage NAs
id                                                 integer   0
fudays                     Follow Up  days        integer   0
status      Original Survival Status             3 integer   0
drug                       Treatment            2 integer  25
age                              Age  days        integer   0
sex                            Gender            2 integer   0
ascites          Presence of Ascites             2 integer  25
bili                 Serum Bilirubin mg/dL         double   0
chol               Serum Cholesterol mg/dL        integer  32
album                 Serum Albumin mg/dL         double   0
stage    Histologic stage of disease             4 integer   2
ageyrs                           Age years         double   0
fuyrs                      Follow Up years         double   0
censored   Collapsed Survival Status             2 integer   0

+--------+-----------------------------------------------+
|Variable|Levels                                         |
+--------+-----------------------------------------------+
|status  |Censored,Censored due to liver treatment,Dead|
+--------+-----------------------------------------------+
|drug    |D-penicillamine,Placebo                        |
+--------+-----------------------------------------------+
|sex     |Female,Male                                    |
+--------+-----------------------------------------------+
|ascites |No,Yes                                         |
+--------+-----------------------------------------------+
|stage   |1,2,3,4                                        |
+--------+-----------------------------------------------+
|censored|Censored,Dead                                  |
+--------+-----------------------------------------------+
```

Like the first document, all R code has been extracted from this document and saved in the `Scott.IntroToR.II.R` text file.

Chapter 1

Working with Data Structures

Learning objective
To understand how to generate, manipulate, and use R's various data structures.

1.1 Vectors

DEFINITION: As mentioned in the first document, the vector is the simplest data structure in R. For example, a single value in R (i.e., the logical value `TRUE` or the numeric value `2`) is actually just a vector of length 1. Vectors are one dimensional and consist of an ordered collection of elements. All elements of a vector must be the same data type, or *mode* – *numeric* (an amalgam of *integer* and *double* precision mode), *complex* (numeric value followed by an `i`), *logical*, *character*, and *raw* (intended to hold raw bytes). However, vectors can include missing elements designated with the `NA` value.

CREATION: We demonstrated in the first document how vectors could be created using the `c()` (concatenate) and `seq()` (sequence) functions. Recall, the `seq()` function constructs a numeric vector, while the `c()` function can be used to generate all kinds of vectors. There are many other functions that will construct a vector, including the `rep()` (replicate), `sample()`, and `paste()` functions. The `rep()` function replicates the specified values to generate a vector. The `sample()` function generates a vector by taking a random sample of the specified values. And the `paste()` function generates character vectors by concatenating the specified elements. The following are examples of these three functions:

```
> rep(1:4, times = c(2, 4, 3, 1))

 [1] 1 1 2 2 2 2 3 3 3 4

> sample(c("M", "F"), size = 10, replace = TRUE)

 [1] "F" "F" "M" "F" "F" "M" "M" "F" "M" "M"

> paste("Treatment", c("A", "B", "C"))

[1] "Treatment A" "Treatment B" "Treatment C"
```

Note, the `set.seed()` function is often used in conjunction with the `sample()` function in order to (re-)generate the same random sample.

There are also several functions in addition to the `factor()` function that will generate factors. These are the `gl()`, `cut()`, and `interaction()` functions. The `gl()` function generates a factor using the specified pattern of its levels. The `cut()` function generates a factor by dividing the range of a numeric vector

into intervals and coding the numeric values according to which interval they fall into. And, as its name implies, the `interaction()` function generates a factor that represents the interaction of the specified levels. The following are examples of these three functions. You'll notice that we use the `levels()` function in conjunction with the `interaction()` function to build a factor from the levels of two existing factors.

```
> gl(n = 2, k = 8, label = c("Control", "Treatment"))

 [1] Control   Control   Control   Control   Control   Control   Control   Control
 [9] Treatment Treatment Treatment Treatment Treatment Treatment Treatment Treatment
Levels: Control Treatment

> head(cut(pbc$ageyrs, breaks = 4))

[1] (54.5,66.5] (30.5,42.5] (54.5,66.5] (42.5,54.5] (30.5,42.5] (54.5,66.5]
Levels: (30.5,42.5] (42.5,54.5] (54.5,66.5] (66.5,78.5]

> interaction(levels(pbc$drug), levels(pbc$censored))

[1] D-penicillamine.Censored Placebo.Dead
4 Levels: D-penicillamine.Censored Placebo.Censored ... Placebo.Dead
```

It is also worthwhile to mention character vectors in a little more detail. Specifically, single or double quotes can be embedded in character strings. For example,

```
> "Single 'quotes' can be embedded in double quotes"

[1] "Single 'quotes' can be embedded in double quotes"
```

Similarly, double quotes can be embedded into single quote character strings (e.g., `'Example of "double" inside single'`), but the single quotes will be converted to double quotes and the embedded double-quotes will be converted to `\"`. For example, compare the code specified in the `Scott.IntroToR.II.R` file for the following expression and that shown as output.

```
> "Double \"quotes\" can be embedded in single quotes"

[1] "Double \"quotes\" can be embedded in single quotes"
```

In actuality, in order to embed double quotes within a character string specified using double quotes, you must *escape* them using a backslash, \. For example,

```
> "Embedded double quotes, \", must be escaped"

[1] "Embedded double quotes, \", must be escaped"
```

This `\"` character specification will make more sense when we discuss functions like as `cat()` and `write.table()` in the 'Data Export' section.

ATTRIBUTES: All vectors have a length, which is its number of elements and is calculated using the `length()` function. For example,

```
> length(pbc$ageyrs)

[1] 100
```

However, the `length()` function does not distinguish missing elements, `NA`s, from the length of a vector. For example, we know that the `ascites` column of our `pbc` data frame contains 25 missing values, yet

```
> length(pbc$ascites)

[1] 100
```

To calculate the number of *non-missing* elements of a vector, we take advantage of an odd quirk of the logical values, `TRUE` and `FALSE`. Specifically, the logical values of `TRUE` and `FALSE` are *coerced* to values of `1` and `0`, respectively, when used in arithmetic expressions. So, we can calculate the number of non-missing elements of a vector using

```
> sum(!is.na(pbc$ascites))

[1] 75
```

An alternative is to tabulate the result of the `is.na()` function using the `table()` function, as demonstrated in the first document:

```
> table(!is.na(pbc$ascites))

FALSE  TRUE
   25    75
```

These same tips can be used with factors.

If the elements of a vector are named, the `names()` function returns (prints) the associated names. For example,

```
> (x <- c(Dog = "Spot", Horse = "Penny", Cat = "Stripes"))

      Dog     Horse       Cat
   "Spot"   "Penny" "Stripes"

> names(x)

[1] "Dog"   "Horse" "Cat"
```

As mentioned in the first document, the `levels()` function can be used to return (print) the levels of a specified factor. Another useful function is the `nlevels()` function, which returns (prints) the number of levels of a specified factor. Some examples:

```
> levels(pbc$stage)

[1] "1" "2" "3" "4"

> nlevels(pbc$stage)

[1] 4
```

In addition, the `mode()` function returns (prints) the mode of the specified vector. Recall, all factors are stored internally as integer vectors – more to come in 'Coercing the mode of a vector'. For example,

```
> mode(x)

[1] "character"

> mode(pbc$ageyrs)

[1] "numeric"

> mode(pbc$drug)

[1] "numeric"
```

SUBSETTING: For a vector, we can use single square brackets, `[ ]`, to extract specific subsets of the vector's elements. Specifically, for a vector `x`, we use the general form `x[index]`, where `index` is a vector that can be one of the following forms:

1. **A vector of positive integers.** The values in the `index` vector normally lie in the set `{1, 2, ..., length(x)}`. The corresponding elements of the vector are extracted, in that order, to form the result. The `index` vector can be of any length and the result is of the same length as the `index` vector. For example, `x[6]` is the sixth element of `x`, `x[1:10]` extracts the first ten elements of `x` (assuming `length(x)` $\geq$ 10), `x[length(x)]` extracts the last element of `x`, and `x[c(1, 5, 20)]` extracts the 1st, 5th, and 20th elements of x (assuming `length(x)` $\geq$ 20). We can use any of the functions that construct numeric vectors to define the `index` vector of positive integers, such as the `c()` (concatenate), the `rep()` (replicate), the `sample()`, and the `seq()` (sequence) functions or the `:` (colon) operator. `NA` is returned if the `index` vector contains an integer > `x`, and an empty vector (`numeric(0)`) is returned if the `index` vector contains a `0`. Alternatives in this situation are the `head()` and `tail()` functions, which will return (print) the first/last `n=` elements of a vector (`n = 6` by default).

2. **A vector of negative integers.** This specifies the values to be *excluded* rather than *included* in the extraction. For example, `x[-c(1:5)]` extracts all but the first five elements of `x` – we merely place a negative sign in front of the `index` vector. As seen, all elements of `x` except those that are specified in the `index` vector are extracted, in their original order, to form the result. The results is the `length(x)` minus the length of the `index` vector elements long.

3. **A logical vector.** In this case, the `index` vector must be of the same length as `x`. Values corresponding to `TRUE` in the `index` vector are extracted and those corresponding to `FALSE` or `NA` are not. The logical `index` vector can be explicitly given (e.g., `x[c(TRUE, FALSE, TRUE)]`) or can be constructed as a *conditional expression* using any of the comparison or logic operators – see the vector 'Manipulation' section for more detail. For example,

```
> set.seed(1)
> (x <- sample(10, size = 10, replace = TRUE))

 [1]  3  4  6 10  3  9 10  7  7  1

> x[x == 4]

[1] 4

> x[x > 2 & x < 5]

[1] 3 4 3
```

   As seen, the result is the same length as the number of `TRUE` values in the `index` vector. Recall, if the vector we are trying to subset contains missing values, we need to make sure our logical indexing vector correctly excludes them using the `is.na()` function, if desired. For example,

```
> (x <- c(9, 5, 12, NA, 2, NA, NA, 1))

[1]  9  5 12 NA  2 NA NA  1

> x[x > 2]

[1]  9  5 12 NA NA NA

> x[x > 2 & !is.na(x)]

[1]  9  5 12
```

   An alternative in this situation is the `subset()` function. You merely specify the vector as its first argument and a conditional expression as its second. An advantage to the `subset()` function is that missing values are automatically taken as false, so they do not need to be explicitly removed using the `is.na()` function. For example,

```
> subset(x, x > 2)

[1]  9  5 12

> subset(x, x > 2 & x < 10)

[1] 9 5
```

4. **A vector of character strings.** This possibility applies only when the elements of a vector have *names*. In that case, a subvector of the names vector may be used in the same way as the positive integers case in 1. That is, the strings in the `index` vector are matched against the names of the elements of `x` and the resulting elements are extracted. Alphanumeric names are often easier to remember than numeric indices of elements. For example,

```
> (fruit <- c(oranges = 5, bananas = 10, apples = 1, peaches = 30))

oranges bananas  apples peaches
      5      10       1      30

> names(fruit)

[1] "oranges" "bananas" "apples"  "peaches"

> fruit[c("apples", "oranges")]

 apples oranges
      1       5
```

IMPORTANT: As hinted to in the 'Assignment' section, extracting elements of a data structure, such as a vector, can be done on the left hand side of an assignment expression (e.g., to select parts of a vector to replace) as well as on the right-hand side. For example, we can assign the missing values of the `x` vector assigned above to zeros:

```
> x

[1]  9  5 12 NA  2 NA NA  1

> x[is.na(x)] <- 0
> x

[1]  9  5 12  0  2  0  0  1
```

Note, the *recycling rule*, discussed in the vector 'Manipulation' section, was used in `x[!is.na(x)] <- 0`. Specifically, the value of 0 was 'recycled' to fill in the three missing values of `x`.

MANIPULATION: Because there are several kinds of vectors, there are many ways in which vectors can be manipulated. In this document, we will discuss (1) how the vectorization of functions and the recycling rule affect vector manipulations; (2) the construction and use of conditional expressions; (3) how to coerce the mode of a vector; (4) how to construct and manipulate vectors of dates; and (5) how character vectors can be manipulated using regular expressions.

VECTORIZATION OF FUNCTIONS & THE RECYCLING RULE: Many R functions are *vectorized*, meaning that they operate on each of the elements of a vector (i.e., *element-by-element*). For example, the expression `log(y)` returns a vector that is the same length of `y`, where each element of the result is the natural logarithm of the corresponding element in `y`.

```
> (y <- sample(1:100, size = 10))
```

```
 [1] 21 18 68 38 74 48 98 93 35 71
```

```
> log(y)
```

```
 [1] 3.044522 2.890372 4.219508 3.637586 4.304065 3.871201 4.584967 4.532599 3.555348
[10] 4.262680
```

Other examples of vectorized functions include the `exp()` (exponentiation) function, the `sqrt()` (square root) function, the `round()` function, and the `casefold()` function.

In addition, any of the arithmetic (mentioned in the first document), comparison, and logic operators (mentioned in the next section) will operate element-by-element. For example, adding two vectors of the same length:

```
> (x <- sample(1:20, size = 10))
```

```
 [1] 19  5 12  3 20  6  1 16 11  4
```

```
> x + y
```

```
 [1]  40  23  80  41  94  54  99 109  46  75
```

These operators and other specific functions incorporate what is known as the '*recycling rule*'. The rule is that shorter vectors in the expression are replicated (i.e., recycled; re-used) to be the length of the longer vector. The evaluated value of the expression is a vector of the same length as the longest vector which occurs in the expression. The simplest illustration of the recycling rule is when the expression involves a 'constant' (e.g., a single numeric value):

```
> y + 2
```

```
 [1]  23  20  70  40  76  50 100  95  37  73
```

```
> y < 50
```

```
 [1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

In each of these examples, the constant, which is technically a vector of length 1, is replicated to the length of `y`.

When the length of the shorter vector is greater than 1, the elements of the shorter vector are replicated *in order* until the result is the proper length. For example,

```
> (x <- sample(1:20, size = 2))
```

```
[1] 10 12
```

```
> x + y
```

```
 [1]  31  30  78  50  84  60 108 105  45  83
```

Here, the vector `x` was replicated 5 times to match the length of `y` (10 elements).

If the length of the longer vector is not a multiple of the shorter one, the shorter vector is *fractionally* replicated and a warning is given (`longer object length is not a multiple of shorter object length`). For example (warning is given, but not shown in output),

```
> x <- 1:9
> y <- 1:10
> x + y
```

```
 [1]  2  4  6  8 10 12 14 16 18 11
```

Here, to match the length of `y` only the first element of `x` (1) was re-used.

CONDITIONAL EXPRESSIONS: When using R, vectors of logical values, `TRUE` and `FALSE`, are most often generated by *conditional expressions*. For example, the expression `y <- x > 10` assigns `y` to be a vector of the same length as `x` with value `FALSE` corresponding to elements of `x` where the condition is *not* met and `TRUE` where it is. Conditional expressions are often constructed using *comparison* and *logic operators*, which are listed in the following table:

Type	Operator	Meaning
Comparison	<	Less than
	>	Greater than
	==	Equal to
	!=	Not equal to
	>=	Greater than or equal to
	<=	Less than or equal to
Logic	&, &&	And (intersection)
	|, ||	Or (union)
	!	Not (negation)

As their name implies and has been demonstrated, the comparison operators are used to compare two vectors – e.g., `x < y`. In addition, as mentioned above, the comparison operators are *vectorized*. Therefore, they will compare each element of the first vector with each element of the second vector, employing the recycling rule when necessary – e.g., comparing a 5 element numeric vector to the value 2. Comparison operators may be applied to vectors of any mode and they return a logical vector of the same length as the input vectors.

In contrast, the logic operators are applied to logical vectors, which are usually the result of comparing two vectors. As shown in the table above, the "And" and "Or" logical operators exist in two forms. The logic operators `&` and `|`, compare each element of the two specified logical vectors (e.g., `x & y`) and return a logical *vector* the same length as the input vectors. In contrast, `&&` and `||` only return a single logical value for the outcome – evaluating only the first element of the two input vectors. The `&&` and `||` forms are most often used in looping expressions, particularly in `if` statement – see the 'Conditional Evaluation of Expressions' section of the second chapter. The following is an example showing the difference

```
> (x <- 1:10)

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

> x < 7 & x > 2

 [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE

> x < 7 && x > 2

[1] FALSE
```

Note, the expression `2 < x < 7` is an invalid alternative in the example above; the expression must be broken up into two comparison expressions combined with a logic operator.

When using logic operators, it helps to be aware of how various expressions will evaluate:

- `TRUE & TRUE = TRUE`

- TRUE & FALSE = FALSE
- FALSE & FALSE = FALSE
- TRUE | TRUE = TRUE
- TRUE | FALSE= TRUE
- FALSE | FALSE = FALSE

How various expressions will evaluate becomes a little more tricky when NAs are involved:

- NA & TRUE = NA
- NA | TRUE = TRUE
- NA & FALSE = FALSE
- NA | FALSE = NA

Other useful operators include the %in% and the (Hmisc package's) %nin% value matching operators. These binary operators return a logical vector indicating whether elements of the vector specified as the left operand match any of the values in the vector specified as the right operand. The %nin% operator is the 'negative' of the %in% operator. For example,

```
> (x <- sample(c("A", "B", "C", "D"), size = 10, replace = TRUE))

 [1] "B" "A" "D" "C" "D" "A" "C" "B" "D" "C"

> x %in% c("A", "C", "D")

 [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE

> x %nin% c("A", "B")

 [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
```

These operators are useful as alternatives to multiple 'Or' statements. For example, `x %in% c("A", "C", "D")` is equivalent to `x == "A" | x == "C" | x == "D"`. Also, to use the %nin% operator, ***DON'T FORGET*** to load the Hmisc package with the `library()` function (i.e., `library(Hmisc)`) if you haven't already done so. If you haven't previously installed the package, you must do this first.

To 'wholly' compare two vectors, two functions are available: the `identical()` and `all.equal()` functions. Some examples:

```
> x <- 1:3
> y <- 1:3
> x == y

[1] TRUE TRUE TRUE

> identical(x, y)

[1] TRUE

> all.equal(x, y)

[1] TRUE
```

The `identical()` function compares the internal representation of the data and returns `TRUE` if the objects are strictly identical, and `FALSE` otherwise. On the other hand, the `all.equal()` function compares the 'near equality' of two objects, and returns `TRUE` or displays a summary of the differences. The 'near equality' arises from the fact that the `all.equal()` function takes the approximation of the computing process into account when comparing numeric values. The comparison of numeric values on a computer is sometimes surprising! For example,

```
> 0.9 == (1 - 0.1)

[1] TRUE

> identical(0.9, 1 - 0.1)

[1] TRUE

> all.equal(0.9, 1 - 0.1)

[1] TRUE

> 0.9 == (1.1 - 0.2)

[1] FALSE

> identical(0.9, 1.1 - 0.2)

[1] FALSE

> all.equal(0.9, 1.1 - 0.2)

[1] TRUE

> all.equal(0.9, 1.1 - 0.2, tolerance = 1e-16)

[1] "Mean relative difference: 1.233581e-16"
```

COERCING THE MODE OF A VECTOR: We've mentioned thus far that logical vectors may be used in arithmetic expressions, in which case they are *coerced* into numeric vectors – `FALSE` becoming `0` and `TRUE` becoming `1`. For example,

```
> x <- c(34, 31, 80, 78, 64, 87)
> x > 35

[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE

> sum(x > 35)

[1] 4

> sum(x > 35)/length(x)

[1] 0.6666667
```

In fact, it is possible to coerce a vector of any mode to any other mode, even though the result may not be what you wish. These coercions are possible using the many `as.X()` vector functions in R – in fact, there are many `as.X()` functions that convert other data structures, but they will only be mentioned in the 'Catalog' chapter. Even though there are many such `as.X()` functions for coercing vectors, I have found that I primarily use just three of them – the `as.numeric()`, the `as.character()`, and `as.logical()` functions, which convert vectors to mode numeric, character, and logical, respectively. Even though the idea of the coercion follows intuitive rules, the results of the three functions depend on the mode of the input vector, as summarized in the following table:

Function	Coercion (Input ⇒ Output)

`as.numeric()`	`FALSE` ⇒ `0`
	`TRUE` ⇒ `1`
	`"1"`, `"2"`, `...` ⇒ `1`, `2`, `...`
	other characters ⇒ `NA`

`as.character()`	`1, 2, ...` ⇒ `"1", "2", ...`
	`FALSE` ⇒ `"FALSE"`
	`TRUE` ⇒ `"TRUE"`

`as.logical()`	`0` ⇒ `FALSE`
	other numbers (e.g., `1`) ⇒ `TRUE`
	`"FALSE"`, `"F"` ⇒ `FALSE`
	`"TRUE"`, `"T"` ⇒ `TRUE`
	other characters ⇒ `NA`

The `as.numeric()` and `as.character()` functions are often used when trying to coerce a *factor* back to a numeric or character vector. When dealing with a factor with character levels, the `as.character()` function will coerce it back to a character vector. For example,

```
> (fac <- factor(sample(c("M", "F"), size = 5, replace = TRUE)))

[1] F F F F M
Levels: F M

> as.character(fac)

[1] "F" "F" "F" "F" "M"
```

This tip is useful considering some functions will return the internal integer vector representation of a factor with character levels instead of the character levels. For example, compare the output of the following two `ifelse()` function expressions – we will discuss the `ifelse()` function in more detail in upcoming sections.

```
> ifelse(fac == "M", NA, fac)

[1]  1  1  1  1 NA

> ifelse(fac == "M", NA, as.character(fac))

[1] "F" "F" "F" "F" NA
```

Coercing a factor with character levels to a numeric vector with the `as.numeric()` function returns (prints) the internal integer vector representation of the factor. For example,

```
> as.numeric(fac)

[1] 1 1 1 1 2
```

In order to coerce a factor with numeric levels into a numeric vector, keeping the levels as they are originally specified, you have to first coerce the factor into a character vector and then into a numeric vector. For example,

```
> (fac <- factor(sample(5:10, size = 10, replace = TRUE), levels = 5:10))

 [1] 7  9  9  7  10 7  6  5  5  6
Levels: 5 6 7 8 9 10

> as.numeric(fac)
```

```
 [1] 3 5 5 3 6 3 2 1 1 2

> as.numeric(as.character(fac))

 [1]  7  9  9  7 10  7  6  5  5  6
```

DATES: We often deal with data in form of a date and/or time, such as a date of birth, procedure, or death. In turn, we often wish to manipulate these date/time fields to calculate values such as the time to death from baseline or the time between treatment visits. Luckily, there are two main 'datetime' classes in R's `base` package – *Date* and *POSIX*. The *Date* class supports dates *without* times. As mentioned in the 'R Help Desk' article of the June 2004 R Newsletter, 'eliminating times simplifies dates substantially since not only are times, themselves, eliminated but the potential complications of time zones and daylight savings time vs. standard time need not be considered either.' The *POSIX* class actually refers to the two classes of `POSIXct` and `POSIXlt`. It also refers to a `POSIXt` class, which is the superclass of `POSIXct` and `POSIXlt`. Normally we will not access the `POSIXt` class directly when using R, but any `POSIXlt` and `POSIXct` class specific methods of generic functions will be given under `POSIXt`. The `POSIXct` and `POSIXlt` classes support times and dates including times zones and standard vs. daylight savings time. `POSIXct` datetimes are represented as the number of seconds since January 1, 1970 GMT, while `POSIXlt` datetimes are represented by a *list* ('lt') of 9 components plus an optional timezone attribute. The `Date` class and `POSIXct`/`POSIXlt` classes have a similar interface, making it easy to move between them.

To construct a `Date` class datetime object, we use the `as.Date()` function; and to construct a `POSIXlt` datetime object, we use the `strptime()` function. To only way to construct a `POSIXct` datetime object is to coerce the output of the `strptime()` function using the `as.POSIXct()` function. The data argument to the `as.Date()` and `strptime()` functions is a *character* vector specifying the dates (and possibly times). Remember, by default, the `read.table()` and `data.frame()` functions coerce character columns, like datetime columns, to factors. Therefore, make sure you specify that theses functions should treat the character datetime columns as character and/or coerce the possibly factor datetime columns back to character columns using the `as.character()` function. The default format of the date is `"year-month-day"`, such as `"2007-09-24"`. If the variable includes time, then the default format is `"year-month-day hour:minutes:seconds"`, where the hours are specified using a 24-hour clock (i.e., 00-23). The second argument to these three functions is a `format=` argument, which is used to specify how the dates (and times) are represented in the character string. The format consists of *conversion specifications* to represent the different portions of the date/time and characters that represent the delimiters between the different date/time portions, like dashes or forward slashes. The following table lists a subset of the conversion specifications the `as.Date()` and `strptime()` functions will recognize – see the `strptime()` function's help file for a complete list.

Portion of date/time field	Conversion specification & description	
Year	%Y	Numeric year with century (e.g., 2007)
	%y	Numeric year without century (00-99); don't use!
Month	%m	Numeric month (01-12)
	%b	Abbreviated month name (first three letters; e.g., Mar)
	%B	Full month name (e.g., March)
Day	%d	Numeric day (01-31)
Hours	%H	Hours from 24-hour clock (00-23)
	%I	Hours from 12-hour clock (01-12)
	%p	AM/PM indicator (used in conjunction with %I)
Minutes	%M	Minutes (00-59)
Seconds	%S	Seconds (00-61)

Note, leading zeros in the conversion specifications (like months and days) are useful to include in the character strings – I often have problems when I don't. Let's demonstrate these various conversion specifications

with some examples – you'll notice the format of the output is consistent:

```
> as.Date("2007-10-18", format = "%Y-%m-%d")

[1] "2007-10-18"

> as.Date("2007OCT18", format = "%Y%b%d")

[1] "2007-10-18"

> as.Date("October 18, 2007", format = "%B %d, %Y")

[1] "2007-10-18"

> strptime("10/18/2007 08:30:45", format = "%m/%d/%Y %H:%M:%S")

[1] "2007-10-18 08:30:45"

> strptime("10/18/2007 12:30:45 AM", format = "%m/%d/%Y %I:%M:%S %p")

[1] "2007-10-18 00:30:45"
```

The `seq()` and `cut()` functions also have `Date` and `POSIXt` class specific methods for generating datetime vectors – see the `seq.Date()`, `seq.POSIXt()`, `cut.Date()`, and `cut.POSIXt()` help files for examples. The `seq()` functions are particularly useful when creating a date axis for a plot. In addition, once a datetime vector has been constructed, you can manipulate it in several ways – see the help files for the `Date` and `POSIXt` methods of the `round()`, `trunc()`, `diff()`, `weekdays()` and `format()` functions. Also check out `?"+.Date"` and `?"+.POSIXt"`, and the `difftime()` help file. The following are some examples:

```
> (x <- seq.Date(from = as.Date("2007-10-18"), to = as.Date("2007-10-30"),
+     by = "3 days"))

[1] "2007-10-18" "2007-10-21" "2007-10-24" "2007-10-27" "2007-10-30"

> x + 10

[1] "2007-10-28" "2007-10-31" "2007-11-03" "2007-11-06" "2007-11-09"

> x > as.Date("2007-10-21")

[1] FALSE FALSE  TRUE  TRUE  TRUE

> x - as.Date(c("2006-01-10", "2007-08-15", "2005-06-24", "2004-12-30",
+     "2005-04-05"))

Time differences in days
[1]  646   67  852 1031  938

> diff(x)

Time differences in days
[1] 3 3 3 3

> weekdays(x)

[1] "Thursday"  "Sunday"    "Wednesday" "Saturday"  "Tuesday"

> format(x, "%Y")

[1] "2007" "2007" "2007" "2007" "2007"
```

```
> format(x, "%m/%d/%Y")

[1] "10/18/2007" "10/21/2007" "10/24/2007" "10/27/2007" "10/30/2007"
```

It is important to mention that you will receive an error if you try to define a new column of a data frame as the output of the `strptime()` function. For example, you would receive an error if you invoked an expression similar to `df$visitdate <- with(df, strptime(visit, format = "%Y-%m-%d"))`, where `df` is a fictitious data frame with `visit` being a character column giving the visit dates. This error will occur because, as we mentioned, the `strptime()` function returns a *list*, even though it does not appear this way (even if we look at the structure of the output using the `str()` function). The remedy to this problem is the above-mentioned `as.POSIXct()` function, which will coerce the `strptime()` function output to a datetime vector. Therefore, we should modify the example code to: `df$visitdate <- with(df, as.POSIXct(strptime(visit, format = "%Y-%m-%d")))`.

In addition to the `POSIXct` and `POSIXlt` classes to represent dates and times, there is also the `chron` package, which is an add-on package available through the CRAN website. The `chron` package provides dates and times, but there are no time zones or notion of daylight vs. standard times. Datetimes in the `chron` package are represented internally as days since January 1, 1970, with times represented as fractions of days. Even though the `chron` package includes several functions that allow you to manipulate datetime variables, the format used to specify the dates and times is not as extensive as the conversion specifications of the `POSIXct` and `POSIXlt` classes.

MANIPULATING CHARACTER VECTORS WITH REGULAR EXPRESSIONS: Working with data in R often involves text data, which are often called character strings. These character strings may be the values of a column in a data frame or may be the output from a function like the `names()` function. In any case, we often want to extract or manipulate elements of character vectors that match a specific pattern of characters. In R, this is possible using the `grep()` and `sub()` functions, respectively, and *regular expressions*. In general, regular expressions are used to match patterns against character strings. In other words, regular expressions are special strings of characters that you create in order to locate matching pieces of text. In order to understand the power of regular expressions, let's work through their use with the `grep()` function, which searches for matches to a `pattern=`, which is specified as its first argument, within the character vector `x=`, which is specified as its second argument. More information is available regarding regular expressions in the `regex` help file (ie, `?regex`).

Patterns: As we have indicated, every regular expression contains a pattern, which is used to match the regular expression against a character string. Within a pattern, all characters except `.`, `|`, `(`, `)`, `[`, `]`, `{`, `}`, `+`, `\`, `^`, `$`, `*`, and `?` match themselves. For example, we could extract the column names of our `pbc` data frame that contain the character string `"age"` using the `grep()` function. Note, we need to also specify `value = TRUE` in our `grep()` function expression to return (print) the matching values – by default, the `grep()` function returns the indices of the matches (`value = FALSE`).

```
> names(pbc)

 [1] "id"       "fudays"   "status"   "drug"     "age"      "sex"      "ascites"
 [8] "bili"     "chol"     "album"    "stage"    "ageyrs"   "fuyrs"    "censored"

> grep(pattern = "age", x = names(pbc), value = TRUE)

[1] "age"    "stage"  "ageyrs"
```

If you want to match one of the special characters mentioned above literally, you have to precede it with *two* backslashes. For example, we could extract the elements of the following character vector that contain a period.

```
> (char <- c("id", "patient.age", "date", "baseline_bmi", "follow.up.visit"))
```

```
[1] "id"              "patient.age"     "date"            "baseline_bmi"
[5] "follow.up.visit"

> grep(pattern = "\\.", x = char, value = TRUE)

[1] "patient.age"     "follow.up.visit"
```

Anchors: By default, a regular expression will try to find the first match for the pattern in a string. But what if you want to force a pattern to match only at the start or end of a character string? In this case, the `^` and `$` special characters are used to match an *empty string* at the beginning and end of a line, respectively. So, for example,

```
> char <- c("this is an option", "or perhaps this", "and don't forget about this one")
> grep(pattern = "this", x = char, value = TRUE)

[1] "this is an option"               "or perhaps this"
[3] "and don't forget about this one"

> grep(pattern = "^this", x = char, value = TRUE)

[1] "this is an option"

> grep(pattern = "this$", x = char, value = TRUE)

[1] "or perhaps this"
```

Character classes: A *character class* is a list of characters between brackets, `[ ]`, which matches any single character between the brackets. On the other hand, if the first character of the list is the caret, `^`, then the character class matches any characters *not* in the list. So, for example, `[aeiou]` matches any vowel and `[^abc]` matches anything except the characters `a`, `b`, or `c`. A range of characters may be specified by giving the first and last characters, separated by a hyphen. So, `[0-9]` matches any digits, `[a-z]` matches any lower case letters, `[A-Z]` matches any upper case letters, `[a-zA-Z]` matches any alphabetic characters, `[^a-zA-Z]` matches any non-alphabetic characters, `[a-zA-Z0-9]` matches any alphanumeric characters, `[ \t\n\r\f\v]` matches any white space characters, and `[,.:;!?]` matches punctuation. Also, the significance of the special characters, `.|()[]{}+^$*?`, is turned off inside the brackets. In addition, to include a literal `[` or `]`, place it anywhere in the list; to include a literal `^`, place it anywhere but first; and to include a literal `-`, place it first or last. To match any other special character, except \, inside a character class, place it anywhere. Here's some examples:

```
> char <- c(" ", "3 times a day")
> grep(pattern = "[a-zA-Z0-9]", x = char, value = TRUE)

[1] "3 times a day"

> grep(pattern = "[^a-zA-Z0-9]", x = char, value = TRUE)

[1] " "             "3 times a day"
```

Repetition: If `r` stands for the immediately preceding regular expression within a pattern, then `r*` matches zero or more occurrences of `r`; `r+` matches one or more occurrences of `r`; and `r?` matches zero or one occurrence of `r`. Additionally, `{n}` matches the preceding item 'n' times; `{n,}` mathces the preceding item 'n' or more times; and `{n,m}` matches the preceding item at least 'n' times, but not more than 'm' times. These repetition constructs have a high precedence – they bind only to the immediately preceding regular expression in the pattern. So, `"ab+"` matches an `a` followed by one or more `b`'s, not a sequence of `ab`'s. You have to be careful with the `*` construct too – the pattern `"a*"` will match any string (i.e., every string has zero or more `a`'s). Here are some examples:

```
> char <- c("The", "moon is made", "of cheese")
> grep(pattern = " +", x = char, value = TRUE)
```

```
[1] "moon is made" "of cheese"

> grep(pattern = "o?o", x = char, value = TRUE)

[1] "moon is made" "of cheese"
```

Alternation: The vertical bar, `|` is a special character because an unescaped vertical bar matches either the regular expression that precedes it or the regular expression that follows it. For example,

```
> char <- c("red", "ball", "blue", "sky")
> grep(pattern = "d|e", x = char, value = TRUE)

[1] "red"  "blue"

> grep(pattern = "al|lu", x = char, value = TRUE)

[1] "ball" "blue"
```

Grouping: You can use parentheses to group terms within a regular expression. Everything written within the group is treated as a single regular expression. For example,

```
> char <- c("red ball", "blue ball", "red sky", "blue sky")
> grep(pattern = "red", x = char, value = TRUE)

[1] "red ball" "red sky"

> grep(pattern = "(red ball)", x = char, value = TRUE)

[1] "red ball"
```

We should also mention what the period, `.`, special character does. An unescaped period, `.`, matches any *single* character. In addition, the `grep()` function is case-sensitive – its `ignore.case=` argument is by default `FALSE`. For example,

```
> char <- c("vit E", "vitamin e")
> grep(pattern = "vit.*E", x = char, value = TRUE)

[1] "vit E"

> grep(pattern = "vit.*E", x = char, value = TRUE, ignore.case = TRUE)

[1] "vit E"     "vitamin e"
```

As I mentioned above, the `sub()` (substitute) function also employs regular expressions. The `sub()` function needs a `pattern=` and `x=` argument like the `grep()` function, but it also needs a `replacement=` argument, which specifies the substitution for the matched pattern. For example,

```
> char <- c("one.period", "two..periods", "three...periods")
> sub(pattern = "\\.+", replacement = ".", x = char)

[1] "one.period"    "two.periods"   "three.periods"
```

With the `sub()` function, the parentheses take on an additional ability. Specifically, the parentheses can be used to tag a portion of the match as a 'variable' to return. For example, we can extract just the leading number from each element of the following character vector.

```
> char <- c("45: Received chemo", "1, Got too sick", "2; Moved to another hospital")
> sub(pattern = "^([0-9]+)[:,;].*$", "\\1", char)

[1] "45" "1"  "2"
```

Here's another example:

```
> char <- c("vit E", "vitamin E", "vitamin ESTER-C", "vit E ")
> sub(pattern = "^(vit).*([E]).*$", "\\1 \\2", char)

[1] "vit E" "vit E" "vit E" "vit E"
```

It is also worthwhile to mention the `gsub()` function. Unlike the `sub()` function, which replaces only the first occurrence of a 'pattern', the `gsub()` function replaces all occurrences. For ezample,

```
> sub(" a", " A", "Capitalizing all words beginning with an a")

[1] "Capitalizing All words beginning with an a"

> gsub(" a", " A", "Capitalizing all words beginning with an a")

[1] "Capitalizing All words beginning with An A"
```

1.2 Matrices & Arrays

DEFINITION: A *matrix* is a two-dimensional data structure that consists of rows and columns – think of a vector with dimensions. Like vectors, all the elements of a matrix must be the same data type (all numeric, character, or logical), and can include missing elements designated with the `NA` value. Taking it one step further, an *array* is a generalization of a matrix, which allows more than two dimensions. In general, an array is k-dimensional.

CREATION: To create a matrix, we can use the `matrix()` function, which constructs an `nrow` x `ncol` matrix from a supplied vector (`data=`).

```
> args(matrix)

function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
NULL
```

By default, the matrix is filled by columns (`byrow = FALSE`), but specifying `byrow = TRUE` fills the matrix by rows. For example,

```
> matrix(1:6, ncol = 3)

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> matrix(1:6, ncol = 3, byrow = TRUE)

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Row and column names can also be specified using the `dimnames=` argument. The `dimnames=` argument is specified as a list of two components giving the row and column names, respectively. For example,

```
> matrix(c(1, 2, 3, 11, 12, 13), nrow = 2, ncol = 3, byrow = TRUE, dimnames = list(c("row1",
+     "row2"), c("C.1", "C.2", "C.3")))

     C.1 C.2 C.3
row1   1   2   3
row2  11  12  13
```

The `cbind()` and `rbind()` functions can also be used to construct a matrix by binding together the data arguments horizontally (column-wise) or vertically (row-wise), respectively. The supplied data arguments can be vectors (of any length) and/or matrices with the same columns size (i.e., the same number of rows) or row size (i.e., the same number of columns), respectively. Any supplied vector is cyclically extended to match the 'length' of the other data arguments if necessary. For example,

```
> cbind(1:3, 7:9)

     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9

> rbind(1:11, 5:15)

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,]    1    2    3    4    5    6    7    8    9    10    11
[2,]    5    6    7    8    9   10   11   12   13    14    15
```

Row and column names are created by supplying the vectors as *named* vectors – i.e., `VECTORname = vector`. For example,

```
> cbind(col1 = 1:3, col2 = 7:9)

     col1 col2
[1,]    1    7
[2,]    2    8
[3,]    3    9

> rbind(row1 = 1:11, row2 = 5:15)

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
row1    1    2    3    4    5    6    7    8    9    10    11
row2    5    6    7    8    9   10   11   12   13    14    15
```

The `array()` function can be used to construct an array. With the `array()` function, two formal arguments must be specified: (1) a vector specifying the elements to fill the array; and (2) a vector specifying the dimensions of the array. For example,

```
> array(1:24, dim = c(3, 4, 2))

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

The array is filled column-wise, and if the data vector is shorter than the number of elements defined by the dimensions vector, the data vector is recycled from its beginning to make up the needed size. Like the

`matrix()` function, dimension names can be added using the `dimnames=` argument.

ATTRIBUTES: The `dim()` function will return (print) the dimensions of the specified matrix or array – the number of rows, columns, etc. If the dimensions of a matrix or array are named (rows, columns, etc), the `dimnames()` function will return (print) these assigned names. Lastly, because matrices/arrays are similar to vectors in the fact that all of their elements must be the same data type, the `mode()` function will return the mode of the matrix/array.

SUBSETTING: Elements (rows and/or columns) of a matrix may be extracted using the single square bracket operators, `[ ]`, by giving *two* index vectors in the form `x[i, j]`, where `i` extracts rows and `j` extracts columns. The index vectors `i` and `j` can take any of the four forms shown for vectors. If character vectors are used as indices, they refer to row or column names, as appropriate. And, if either index vector is not specified (i.e., left empty), then the range of that subscript is taken. That is `x[i, ]` extracts the rows specified in `i` across all columns of `x`, and `x[ , j]` extracts the columns specified in `j` across all rows of `x`. Some examples:

```
> (x <- matrix(1:12, ncol = 4, byrow = TRUE))

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

> x[1, ]

[1] 1 2 3 4

> x[, 4]

[1]  4  8 12

> x[2, 3]

[1] 7

> x[-(1:2), 3:4]

[1] 11 12

> x[x[, 2] < 8 & x[, 4] < 10, ]

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

You can also use the `head()` and `tail()` functions to return (print) the first/last `n=` *rows* a matrix (`n = 6` by default).

Remember, arrays are k-dimensional generalizations of matrices. Therefore, for a k-dimensional array we must give `k` index vectors, each in one of the four forms – `x[i, j, k, ...]`. As with matrices, if any index position is given an empty index vector, then the full range of that subscript is extracted. Some examples:

```
> (x <- array(1:24, dim = c(3, 4, 2)))

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
```

```
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

```
> x[2, 3, 1]

[1] 8

> x[1:2, c(1, 4), -1]

     [,1] [,2]
[1,]   13   22
[2,]   14   23
```

It might not be apparent from the examples above, but sometimes an indexing operation causes one of the dimensions to be 'dropped' from the result. For example, when we return only the fourth column a 3x4 matrix, R returns a three element *vector*, not a one-column matrix.

```
> x <- matrix(1:12, ncol = 4, byrow = TRUE)
> x[, 4]

[1]  4  8 12

> is.matrix(x[, 4])

[1] FALSE

> is.vector(x[, 4])

[1] TRUE
```

The default behavior of R is to return an object of the lowest dimension possible. However, as you can imagine, this isn't always desirable and can cause a failure in general subroutines where an index occasionally, but not usually, has length one. Luckily, this habit can be turned off by adding `drop = FALSE` to the indexing operation. Note, `drop = TRUE` does not add to the index count. For example,

```
> x <- matrix(1:12, ncol = 4, byrow = TRUE)
> x[, 4, drop = FALSE]

     [,1]
[1,]    4
[2,]    8
[3,]   12

> is.matrix(x[, 4, drop = FALSE])

[1] TRUE
```

MANIPULATION: The manipulation of matrices and arrays most often involves mathematical manipulation related to matrix algebra. There are several functions that can be used to manipulate matrices and arrays, including the `t()` (transpose), `aperm()`, `diag()`, `lower.tri()`, `upper.tri()`, `rowsum()`/`colsum()`, `rowSums()`/`colSums()`, `rowMeans()`/`colMeans()`, `crossprod()`, `det()`, `eigen()`, `max.col()`, `scale()`, `svd()`, `solve()`, and `backsolve()` functions. There are additionally two operators that can be used to manipulate matrices and arrays: `%*%` (matrix multiplication) and `%o%` (outer product).

1.3 Data frames & Lists

DEFINITION: As defined in the first document, a *data frame* in R corresponds to what other statistical packages call a 'data matrix' or a 'data set' – the 2-dimensional data structure used to store a complete set of data, which consists of a set of variables (columns) observed on a number of cases (rows). In other words, a data frame is a generalization of a matrix. Specifically, the different columns of a data frame may be of different data types (numeric, character, or logical), but all the elements of any one column must be of the same data type. As with vectors, the elements of any one column can also include missing elements designated with the `NA` value. Most types of data you will want to read into R and analyze are best described by data frames.

A *list* is the most general data structure in R and has the ability to combine a collection of objects into a larger composite object. Formally, a list in R is a data structure consisting of an ordered collection of objects known as its *components*. Each component can contain any type of R object and the components need not be of the same type. Specifically, a list may contain vectors of several different data types and lengths, matrices or more general arrays, data frames, functions, and/or other lists. Because of this, a list provides a convenient way to return the results of a computation – in fact, the results that many R functions return are structured as a list, such as the result of fitting a regression model.

It is also important to realize that a data frame is a special case of a list. In fact, a data frame is nothing more than a list whose components are vectors of the same length and are related in such a way that data in the same 'position' come from the same experimental unit (subject, animal, etc.).

CREATION: As demonstrated in the first document, the `data.frame()` function can be used to construct a data frame from scratch. Often, the data arguments you supply to the `data.frame()` function will be individual vectors, which will construct the columns of the data frame. These data arguments can be specified with or without a corresponding column name – either in the form `value` or the form `COLname = value`. For example, we can use the `sample()` function to generate a random data frame.

```
> ourdf <- data.frame(id = 101:110, sex = sample(c("M", "F"), size = 10,
+     replace = TRUE), age = sample(20:50, size = 10, replace = TRUE),
+     tx = sample(c("Drug", "Placebo"), size = 10, replace = TRUE), diabetes = sample(c(TRUE,
+         FALSE)))
> ourdf

    id sex age      tx diabetes
1  101   F  43    Drug     TRUE
2  102   F  22 Placebo    FALSE
3  103   M  47 Placebo     TRUE
4  104   F  30    Drug    FALSE
5  105   M  46 Placebo     TRUE
6  106   M  30    Drug    FALSE
7  107   M  30    Drug     TRUE
8  108   F  34 Placebo    FALSE
9  109   M  47    Drug     TRUE
10 110   M  46 Placebo    FALSE
```

With the `data.frame()` function, character vectors are automatically coerced into *factors* because of its `stringsAsFactors=` argument. However, specifying `stringsAsFactors = FALSE` in your `data.frame()` function invocation will keep character vectors from being coerced. An alternative is to wrap the character vector with the `I()` function, which keeps it 'as is'. Also, all invalid characters in column names (e.g., spaces, dashes, or question marks) are converted to periods (`.`).

The `list()` function can be used to construct a *list*. Like the `data.frame()` function, the components of a list can be named using the `COMPONENTname = component` construct. For example,

```
> (ourlist <- list(comp1 = c(TRUE, FALSE), comp2 = 1:4, comp3 = matrix(1:20,
+     nrow = 2, byrow = TRUE)))

$comp1
[1]  TRUE FALSE

$comp2
[1] 1 2 3 4

$comp3
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]   11   12   13   14   15   16   17   18   19    20
```

ATTRIBUTES: Also as demonstrated in the first document, we can use the `dim()` function to check the dimensions (number of rows and number of columns, respectively) of a data frame. We can use the `names()` function to check the variable names of a data frame. We can also use the `Hmisc` package's `contents()` function to display both of these attributes and more. Specifically, the `contents()` function displays the *meta-data* of your data frame, which includes the number of observations (rows) and columns, the variable names, the variable labels (if any), the variable units of measurement (if any), the number of levels for *factor* variables (if any), the storage mode of each variable, and the number of missing values (`NA`s) for each variable. The `contents()` function also displays the maximum number of `NA`s across all variables, and the level labels of each *factor* variable (if any).

For a list whose components are named, the `names()` function will return (print) the names of the components. However, the `dim()` and `Hmisc` package's `contents()` function do not work with lists. Instead, use the `length()` function to return (print) the number the number of components of a list. I also recommend using the `str()` (structure) function, which compactly displays the internal structure of an R object. The `str()` function is especially well suited to compactly display the (abbreviated) contents of lists, including nested lists, which are lists with list components. For example, we can return (print) the structure of the `ourlist` list we assigned above.

```
> str(ourlist)

List of 3
 $ comp1: logi [1:2] TRUE FALSE
 $ comp2: int [1:4] 1 2 3 4
 $ comp3: int [1:2, 1:10] 1 11 2 12 3 13 4 14 5 15 ...
```

The output of the `str()` function indicates the `ourlist` is a list of 3 named components – `comp1`, `comp2`, and `comp3`. It also tells us that `comp1` is a logical vector of 2 values (`logi [1:2]`), `comp2` is a numeric vector of 4 values (`int [1:4]`), and `comp3` is a numeric 2x10 matrix (`int [1:2, 1:10]`). The usefulness of the `str()` function will become more apparent when you use it to determine the structure of the output of a function in order to select specific portions of the output – for example, examine the structure of the output from `str(contents(pbc))`.

SUBSETTING: To subset a list, we use either the single square brackets, `[ ]`, or the double square brackets, `[[ ]]`. With the single square brackets, we indicate which *component(s)* of the list we would like to extract. When the components of a list are not named, we specify the desired component(s) using their number(s). For example, if we had a 9 component list `Lst` we could extract the second component using `Lst[2]`. We can also incorporate the colon operator (`:`) or the `c()` (concatenate) function to extract more than one component. For example, we could extract the first three components using `Lst[1:3]` and we could extract the third, fifth, and ninth components using `Lst[c(3, 5, 9)]`. When the components of a list are named, as with the list `ourlist`, we specify the desired component(s) using their name(s) – as quoted character strings. As before, we can incorporate the `c()` (concatenate) function to extract more the one

named component. For example, we could extract the first component of `ourlist` using `ourlist["comp1"]` and we could extract the first and third components using `ourlist[c("comp1", "comp3")]`.

IMPORTANT: Subsetting a list using the single square brackets, `[]`, returns a *list*! Therefore, the result is a sublist of the original list, consisting of the specified components. If it was a named list, the names are transferred to the sublist. For example,

```
> str(ourlist["comp1"])

List of 1
 $ comp1: logi [1:2] TRUE FALSE
```

In contrast, subsetting a list using the double square brackets, `[[ ]]`, extracts the *object* that was stored in the specified component. Because of this, we specify only a single component to be extracted with the double square brackets – we do not incorporate the colon operator, `:`, or the `c()` (concatenate) function. Also, the name of the object is not included when it is extracted, if the corresponding component was named in the list. As with the single square brackets, `[ ]`, the number of the component can be specified using its number or its name, if appropriate. If the components of a list are named, an alternative to the double square brackets is the `$` operator. Therefore, `Lst[["ComponentName"]]` is equivalent to `Lst$ComponentName`. For example,

```
> ourlist[["comp1"]]

[1]  TRUE FALSE

> str(ourlist[["comp1"]])

 logi [1:2] TRUE FALSE

> ourlist$comp3

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]   11   12   13   14   15   16   17   18   19    20

> str(ourlist$comp3)

 int [1:2, 1:10] 1 11 2 12 3 13 4 14 5 15 ...
```

Because the result of subsetting a list using the double square brackets is the stored object, the result can then be itself subsetted as demonstrated for vectors, matrices, etc. For example, we can extract the third column of the `comp3` matrix from `ourlist` using `ourlist$comp3[, 3]`.

Because data frames are a special case of lists, elements (rows and/or columns) of a data frame can be extracted using the `[ ]`, `[[ ]]`, and/or `$` operators. In addition, because a data frame can be thought of as a generalization of a matrix, we can also use the `[ , ]` subsetting format. If `df` is our data frame and `v1`, `v2`, and `v3` are its columns, then `df[["v1"]]`, `df$v1`, and `df[, "v1"]` are equivalent and they all return a vector. However, `df["v1"]` returns a data frame. Similarly, specifying more than one column with the `[ , ]` subsetting format returns a data frame. Some examples using the `ourdf` data frame:

```
> ourdf[, 4]

 [1] Drug    Placebo Placebo Drug    Placebo Drug    Drug    Placebo Drug    Placebo
Levels: Drug Placebo

> ourdf[["tx"]]

 [1] Drug    Placebo Placebo Drug    Placebo Drug    Drug    Placebo Drug    Placebo
Levels: Drug Placebo
```

```
> ourdf$tx

 [1] Drug    Placebo Placebo Drug    Placebo Drug    Drug    Placebo Drug    Placebo
Levels: Drug Placebo

> ourdf[ourdf$sex == "M", c("id", "tx")]

    id      tx
3  103 Placebo
5  105 Placebo
6  106    Drug
7  107    Drug
9  109    Drug
10 110 Placebo
```

Notice how we had to use the $ operator in order to subset the rows of `ourdf` by the `sex` column. ***REMEMBER:*** If the column(s) of the data frame you are using in the conditional selection of rows contain(s) missing values, you will need to explicitly remove the `NA`s. For example, if the `sex` column of `ourdf` contained `NA`s, in order to extract the rows for which `sex == "M"`, we would have to specify `ourdf[ ourdf$sex == "M" & !is.na(ourdf$sex), ]`.

Even though the `dfname$colname` and `dfname[["colname"]]` constructs for extracting a column from a data frame are equivalent, the `dfname$colname` construct is more convenient when using R interactively and the `dfname[["colname"]]` construct is very useful when you are specifying the value of `"colname"` using another object. For example,

```
> x <- "ageyrs"
> head(pbc[[x]])

Age [years]
[1] 66.25873 42.50787 59.95346 52.02464 41.38535 61.72758
```

This `dfname[["colname"]]` construct will become very handy when using loops – see the 'Repetitive Evaluation of Expressions' section of the second chapter.

As you can imagine, using the `[ ]` operators with `dfname$colname` construct and `is.na()` function to properly remove any missing values can lead to a large amount of typing depending on you are trying to subset your data frame. For example, we would have to use the following code to correctly subset the age (in years) values of those female subject who died and had received D-penicillamine:

```
pbc[pbc$sex == "Female" & !is.na(pbc$sex) &
   pbc$censored == "Dead" & !is.na(pbc$censored) &
   pbc$drug == "D-penicillamine" & !is.na(pbc$drug), "ageyrs"]
```

Luckily there are several functions that make subsetting data frames much easier. Let's first discuss the `subset()` function, which we've already introduced with vectors. With data frames, the `subset()` function returns (prints) the specified subset of rows and/or columns of a data frame. The formal arguments of the data frame method of the `subset()` function and their defaults (if any) are:

```
> args(subset.data.frame)

function (x, subset, select, drop = FALSE, ...)
NULL
```

The `x=` argument specifies the data frame to be subsetted. The `subset=` argument specifies which rows of the data frame to keep by using a *conditional* expression. With the `subset=` argument, missing values are automatically taken as false, so they do not need to be explicitly removed using the `is.na()` function.

In addition, the `subset=` argument is evaluated in the data frame, so columns can be simply referred to (by name; e.g, `age`) as variables in the expression. This saves us from having to use the `dfname$colname` construct throughout the specified logical expression. The `select=` argument specifies which columns of the data frames to select. Like the `subset=` argument, for the `select=` argument, we specify the columns by their name. The `select=` argument works by first replacing the column names in the selection expression with the corresponding column numbers in the data frame and then using the resulting integer vector to index the columns. This allows the use of the standard indexing conventions so that, for example, a group of columns can be specified using the `c()` (concatenate) function, ranges of columns can be specified using the `:` operator, or single columns can be dropped using the (unary) `-` operator. Let's demonstrate the usefulness of the `subset()` function with some examples.

```
> subset(pbc, subset = sex == "Female" & censored == "Dead" & drug ==
+     "D-penicillamine" & ascites == "Yes", select = age)

     age
6  22546
18 28018
20 25023

> subset(pbc, subset = sex == "Female" & censored == "Dead" & drug ==
+     "D-penicillamine" & ascites == "Yes", select = c(id, bili, stage))

    id bili stage
6   37  7.1     4
18  92  1.4     4
20 106  2.1     2

> subset(pbc, subset = sex == "Female" & censored == "Dead" & drug ==
+     "D-penicillamine" & ascites == "Yes", select = bili:stage)

   bili chol album stage
6   7.1  334  3.01     4
18  1.4  206  3.13     4
20  2.1   NA  3.90     2

> head(subset(pbc, select = -c(drug, censored, ascites)))

  id fudays status   age    sex bili chol album stage   ageyrs     fuyrs
1  6   2503   Dead 24201 Female  0.8  248  3.98     3 66.25873 6.8528405
2  9   2400   Dead 15526 Female  3.2  562  3.08     2 42.50787 6.5708419
3 20   1356   Dead 21898 Female  5.1  374  3.51     4 59.95346 3.7125257
4 26   1444   Dead 19002 Female  5.2 1128  3.68     3 52.02464 3.9534565
5 30    321   Dead 15116 Female  3.6  260  2.54     4 41.38535 0.8788501
6 37    223   Dead 22546 Female  7.1  334  3.01     4 61.72758 0.6105407
```

Unlike the `[ ]` and `[[ ]]` operators, the `subset()` function *always* returns a data frame, even if the data frame has only one row or one column. To return a single vector of output, we can use the `dfname$colname` construct in conjunction with the `subset()` function. For example,

```
> subset(pbc, subset = sex == "Female" & censored == "Dead" & drug ==
+     "D-penicillamine" & ascites == "Yes")$age

Age [days]
[1] 22546 28018 25023
```

For a data frame, the `complete.cases()` function returns (prints) a logical vector indicating which cases (i.e., rows) of the data frame are 'complete' (i.e., have no missing values across all of its columns). The

`complete.cases()` function can be used in conjunction with the `[ ]` operators, or the `subset()` function. For example, `pbc[complete.cases(pbc), ]` or `subset(pbc, subset = complete.cases(pbc))`, respectively.

The `unique()` function is also useful for subsetting a data frame. In particular, it is very useful when you are trying to determine the possible combinations between several columns that exist in your data frame. For a data frame, the `unique()` function returns a subset of the data frame with all duplicate `rows` (across all columns of the data frame) removed. For example,

```
> unique(subset(pbc, select = c(drug, censored, ascites)))
```

```
              drug censored ascites
1          Placebo     Dead      No
2  D-penicillamine     Dead      No
6  D-penicillamine     Dead     Yes
8          Placebo Censored      No
11 D-penicillamine Censored      No
42         Placebo     Dead     Yes
54 D-penicillamine Censored     Yes
76             <NA> Censored    <NA>
80             <NA>     Dead    <NA>
```

As you noticed, missing values are considered unique values if they exist.

And remember, you can use the `head()` and `tail()` functions to return (print) the first/last `n=` rows of your data frame (`n = 6` by default).

MANIPULATION: In addition to modifying individual variables of a single data frame, you often need to manipulate one or more whole data frames. This data frame manipulation can include tasks such as combining two or more data frames together; sorting the rows of a data frame in ascending or descending order of a desired column or columns; and/or reshaping a data frame that has repeated measurements. We will discuss each of these mentioned data frame manipulations individually.

DEFINING ADDITIONAL INDIVIDUAL COLUMNS: Additional individual columns can be added to an existing data frame in several different ways. The first way is to use the `dfname$colname` construct to assign a value to a new column. For example, we could define follow up in months using `pbc$fumonths <- pbc$fudays/30`. We can also add a new variable by using the `data.frame()` function and overwriting the existing data frame. For example, `pbc <- data.frame(pbc, fumonths = pbc$fudays/30)`. You noticed that we still needed to use the `dfname$colname` construct within the `data.frame()` function invocation to properly reference the needed columns. Yet another way we can add a new variable to an existing data frame is using an assignment expression involving the `transform()` function. By default, the `transform()` function only prints the updated data frame and does not 'permanently' add the new variable to the data frame, so we need to assign the output of the `transform()` function to our data frame. For example, `pbc <- transform(pbc, fumonths = fudays/30)`. An advantage to using `transform()` function to add a variable to a data frame is that you don't have to use the `dfname$colname` construct to reference other columns. Also, don't forget about the `Hmisc` package's `upData()` function, which we demonstrated in the first document. At this point, you might also be asking yourself 'Can't we use the `with()` function to add a new variable to an existing data frame?' The answer is 'no' if we use an expression similar to `with(pbc, newvar <- oldvar*2)`, but we an expression of the form `pbc$newvar <- with(pbc, oldvar*2)` will work. A newer alternative is the `within()` function – `within(pbc, newvar <- oldvar*2)`.

When defining additional individual columns to an existing data frame, the columns can be derived from other existing columns in the data frame, or can be created 'from scratch.' In either case, the column can be constructed by virtually any combination of functions (and operators) in R that construct and/or manipulate

vectors – see the 'Catalog' chapter.

REMOVING COLUMNS/ROWS: A single column can be removed from a data frame by setting it to `NULL`. For example, we could remove `fudays` column from our `pbc` data frame using `pbc$fudays <- NULL`. In a similar sense, we can use the `[ ]` operators to remove several columns from a data frame. For example, `pbc[, c("fudays", "album", "ascites")] <- NULL`. You can check that the column(s) has/have been removed from the data frame by returning (printing) the output of the `names()` function. We can also use the assignment operator (`<-`) and `NULL` to remove a row or multiple rows from a data frame. We would simply use the `[ , ]` subsetting format or the `subset()` function to extract the rows we wish to remove and then assign them to `NULL`. For example, we could remove the female subjects from our `pbc` data frame using `subset(pbc, sex == "Female") <- NULL`.

MERGING: The `cbind()` and `rbind()` functions, that were mentioned in the 'Matrices & Arrays' section, can be used to combine multiple data frames. Recall, in general, the `cbind()` and `rbind()` bind together the data arguments horizontally (column-wise) or vertically (row-wise), respectively. In terms of data frames, the `cbind()` function adds columns of the same length to a data frame, and the `rbind()` function adds (concatenates) rows across the same columns. When combining data frames, the data frames must have the same number of rows in order to `cbind()` them, or the same number of columns *with the same column names* to `rbind()` them. Also, when you use the `cbind()` function to column bind two data frames, you need to make sure that the rows of the data frames are in the same *order*.

An alternative to the `rbind()` and `cbind()` functions is the `merge()` function, which allows more general combinations of data frames. With the `merge()` function, two data frames can be joined in a one-to-one, many-to-many, or many-to-many fashion using any number of matching variables. The formal arguments of the data frame method of `merge()` function and their default values (if any) are:

```
> args(merge.data.frame)

function (x, y, by = intersect(names(x), names(y)), by.x = by,
    by.y = by, all = FALSE, all.x = all, all.y = all, sort = TRUE,
    suffixes = c(".x", ".y"), incomparables = NULL, ...)
NULL
```

The `x=` and `y=` arguments specify the names of the two data frames you wish merge. The `by=`, `by.x=`, and `by.y=` arguments specify the column(s) you wish to merge on. You can use the `by=` argument to specify the column(s) if the name(s) of the column(s) is/are the same in both data frames. For example, if we wanted to merge two data frames on a subject ID column that was named `id` in both data frames, then we would specify `by = "id"`. If the name(s) of the column(s) you wish to merge on are different in the two data frames, then you need to specify the correct names for each data frame using the `by.x=` and `by.y=` arguments. For example, if we wanted to merge two data frames on a subject ID column that was named `id` in the first data frame and `subject` in the second, then we would specify `by.x = "id", by.y = "subject"`. As I've been hinting, you can also merge on more than one column. In this case, use the `c()` (concatenate) function to specify the vector of column names you wish to merge on. For example, `by = c("id", "visitdate")`.

The `all=`, `all.x=`, and `all.y=` arguments specify which subsets of rows should remain in the resulting merged data frames. By default, only those rows 'in common' (i.e., the *intersection*) between the two data frames are returned (`all = FALSE`). In other words, any nonmatching rows of either the first or second data frame are dropped. Use `all = TRUE` to keep all of the rows (matching or nonmatching) from either the first or second data frame (i.e., the *union*). And use `all.x = TRUE` to keep all the nonmatching rows of the first data frame, but drop any nonmatching rows of the second data frame. The use is analogous for `all.y=`. When `all = TRUE`, `all.x = TRUE`, or `all.y = TRUE`, missing values (`NA`s) are filled in the corresponding columns of all of the nonmatching rows.

Also, by default, if the remaining columns in the two data frames (i.e., those columns the data frames contain but were not merged on) have any common names, these have suffixes (by default, '`.x`' and '`.y`') appended

to their names to make the names of the result unique. Use the `suffixes=` argument to change this default.

Let's work through some examples using the `authors` and `books` data frames that are constructed in the `Examples` section of the `merge()` function's help file.

```
> (authors <- data.frame(surname = c("Tukey", "Venables", "Tierney", "Ripley",
+     "McNeil"), nationality = c("US", "Australia", "US", "UK", "Australia"),
+     deceased = c("yes", rep("no", 4))))

   surname nationality deceased
1    Tukey          US      yes
2 Venables   Australia       no
3  Tierney          US       no
4   Ripley          UK       no
5   McNeil   Australia       no

> (books <- data.frame(name = c("Tukey", "Venables", "Tierney", "Ripley",
+     "Ripley", "McNeil", "R Core"), title = c("Exploratory Data Analysis",
+     "Modern Applied Statistics", "LISP-STAT", "Spatial Statistics",
+     "Stochastic Simulation", "Interactive Data Analysis", "An Introduction to R"),
+     other.author = c(NA, "Ripley", NA, NA, NA, NA, "Venables & Smith")))

      name                     title     other.author
1    Tukey Exploratory Data Analysis             <NA>
2 Venables Modern Applied Statistics           Ripley
3  Tierney                 LISP-STAT             <NA>
4   Ripley        Spatial Statistics             <NA>
5   Ripley     Stochastic Simulation             <NA>
6   McNeil Interactive Data Analysis             <NA>
7   R Core      An Introduction to R Venables & Smith

> merge(authors, books, by.x = "surname", by.y = "name")

   surname nationality deceased                     title other.author
1   McNeil   Australia       no Interactive Data Analysis         <NA>
2   Ripley          UK       no        Spatial Statistics         <NA>
3   Ripley          UK       no     Stochastic Simulation         <NA>
4  Tierney          US       no                 LISP-STAT         <NA>
5    Tukey          US      yes Exploratory Data Analysis         <NA>
6 Venables   Australia       no Modern Applied Statistics       Ripley

> merge(books, authors, by.x = "name", by.y = "surname", all.x = TRUE)

      name                     title     other.author nationality deceased
1   McNeil Interactive Data Analysis             <NA>   Australia       no
2   R Core      An Introduction to R Venables & Smith        <NA>     <NA>
3   Ripley        Spatial Statistics             <NA>          UK       no
4   Ripley     Stochastic Simulation             <NA>          UK       no
5  Tierney                 LISP-STAT             <NA>          US       no
6    Tukey Exploratory Data Analysis             <NA>          US      yes
7 Venables Modern Applied Statistics           Ripley   Australia       no

> merge(authors, books, by.x = "surname", by.y = "name", all = TRUE)

   surname nationality deceased                     title     other.author
1   McNeil   Australia       no Interactive Data Analysis             <NA>
```

```
2   Ripley          UK       no        Spatial Statistics             <NA>
3   Ripley          UK       no     Stochastic Simulation             <NA>
4  Tierney          US       no                  LISP-STAT            <NA>
5    Tukey          US      yes Exploratory Data Analysis             <NA>
6 Venables   Australia       no Modern Applied Statistics           Ripley
7   R Core        <NA>     <NA>      An Introduction to R Venables & Smith
```

In all three `merge()` function examples, we had to specify `by.x=` and `by.y=` because the column names were not the same in the two data frames (i.e., `surname` and `name`). In the first merge, all nonmatching rows of `books` were dropped from the result (i.e., the row corresponding to `name` equal `R Core`). You'll also notice that a second row for `surname` equal `Ripley` was added to the result to match the two rows for `name` equal `Ripley` in the `books` data frame. In the second merge, we specified that the nonmatching rows of the `books` data frame should not be dropped (i.e., the `name` equal `R Core` row), and missing values were filled into the remaining columns of the result (`nationality` and `decreased`), which came from the `authors` data frame. In the third merge, all rows from both data frames were kept in the result, and missing values were filled into the appropriate columns – the `nationality` and `deceased` columns for the `surname` equal `R Core` row.

SORTING: Even though the `sort()` function is available to sort the elements of individual vectors, when dealing with data frames, sorting a single vector is not usually what is required. More often, you need to sort a series of columns according to the values of some *other* columns. For example, we may want to sort the `pbc` data frame by treatment (`drug`), gender (`sex`), and survival status (`censored`). With data frames, we also have to make sure that the *whole records* (single rows across all the columns) are correctly sorted, not just individual columns. To sort data frames, we will want to use the `order()` function instead of the `sort()` function. Unlike the `sort()` function, the `order()` function can operate on more than one vector simultaneously. For example, `order(x, y)` will return (print) the index vector which rearranges `x` (its first argument) into ascending or descending order, with ties broken by `y` (its second argument). If more than two arguments are given, ties are broken by subsequent arguments. By default, all vectors involved in the `order()` expression are sorted in ascending order. Because the `order()` function returns an *index vector* and not the original data (like the `sort()`) function), we use the `order()` function in conjunction with the `[ ]` operators to properly sort the rows of a data frame. So, returning to our example, we can sort the `pbc` data frame by treatment (`drug`), gender (`sex`), and survival status (`censored`) using `pbc[with(pbc, order(drug, sex, censored)), ]`. We used the `with()` function to save some typing. When our `order()` expression involves numeric vectors, we can use the (unary) `-` operator to specify that a numeric vector should be sorted in descending order. For example, we can sort the `pbc` data frame by treatment (`drug`), *descending* `age`, and survival status (`censored`) using `pbc[with(pbc, order(drug, - age, censored)), ]`.

RESHAPING: Often, your data frame contains repeated measures. For example, perhaps each subject in your data frame received various kinds of chemotherapy at different visit dates. The `reshape()` function was created to easily manipulate the 'shape' of such longitudinal data frames. For the `reshape()` function, the shape of a data frame can be either 'wide' or 'long.' A 'wide' longitudinal data frame will have one record for each individual, time-constant variables occupying single columns, and time-varying variables occupying a column for each time point. In a 'long' format, the data frame will have multiple records for each individual, with some variables being constant across these multiple records and others varying across these multiple records. A 'long' format data frame also needs a 'time' variable identifying which time point each record comes from and an 'id' variable showing which records refer to the same subject.

So, as its name implies, the `reshape()` function reshapes data frames between the 'wide' and 'long' formats. The formal arguments of the `reshape()` function and their default values (if any) are:

```
> args(reshape)

function (data, varying = NULL, v.names = NULL, timevar = "time",
    idvar = "id", ids = 1:NROW(data), times = seq_along(varying[[1]]),
    drop = NULL, direction, new.row.names = NULL, sep = ".",
```

```
    split = if (sep == "") {
        list(regexp = "[A-Za-z][0-9]", include = TRUE)
    } else {
        list(regexp = sep, include = FALSE, fixed = TRUE)
    })
NULL
```

Other than the `data=` argument, which specifies the name of the data frame you are wanting to reshape and the `drop=` argument, which specifies a vector of column names to drop before reshaping, the remaining arguments of the `reshape()` function are easier to describe depending on which way you are wanting to reshape the data frame – from long to wide, or from wide to long. As you can guess, the `direction=` argument specifies the direction in which the data frame should be reshaped. Use `direction = "wide"` to reshape a long data frame to a wide data frame. And use `direction = "long"` to reshape a wide data frame to a long data frame.

Let's first discuss how to specify the arguments when `direction = "wide"`. The `v.names=` argument specifies the names of the variables in the current long format that will correspond to multiple variables in the resulting wide format. The `idvar=` argument specifies the name(s) of one or more variables in the current long format that identify multiple records from the same group and/or individual. The `timevar=` argument specifies the variable in the long format that is time-varying (i.e., that differentiates multiple records from the same group or individual).

With this in mind, let's work through some `direction = "wide"` examples. For the first example, let's create a data frame with multiple records per individual (i.e., in long format) that has three columns: (1) subject ID; (2) the type of chemotherapy the subject received; and (3) the corresponding visit date. Therefore, each subject can receive different chemotherapy regiments on different visit dates.

```
> x <- data.frame(id = sample(1:100, size = 50, replace = TRUE), chemo = sample(Cs(Hormonal,
+     Antibody, Other), size = 50, replace = TRUE), visitdt = sample(paste(1:12,
+     1:30, 2004:2006, sep = "/"), size = 50, replace = TRUE))
> dim(x)

[1] 50  3

> head(x)

  id    chemo    visitdt
1 15 Antibody  5/17/2005
2 24    Other 10/22/2004
3  6 Antibody 12/12/2006
4 65 Hormonal   4/4/2004
5 88 Hormonal  4/28/2004
6 78    Other   9/9/2006

> table(x$id)

  2   4   6   8  11  13  15  18  21  22  23  24  28  36  42  44  45  46  48  49  50  52
  1   1   1   1   1   1   2   1   1   1   1   1   1   2   1   1   1   2   1   1   1   1
 53  57  58  60  61  64  65  66  72  74  76  78  80  82  88  93  98  99 100
  1   1   1   3   1   1   3   1   1   1   1   1   1   1   1   2   1   1   2

> length(unique(x$id))

[1] 41
```

We want to reshape the data frame from long format to wide format such that there are four columns in the resulting reshaped (wide) data frame: (1) subject ID; (2) 'Hormonal' chemotherapy; (3) 'Antibody' chemotherapy; and (4) 'Other' chemotherapy. In other words, we're interested in tallying whether each subject ever received each of the three types of chemotherapy.

```
> widechemo <- reshape(subset(x, select = Cs(id, chemo)), v.names = "chemo",
+     idvar = "id", timevar = "chemo", direction = "wide")
> head(widechemo)
```

```
  id chemo.Antibody chemo.Other chemo.Hormonal
1 15       Antibody       Other           <NA>
2 24           <NA>       Other           <NA>
3  6       Antibody        <NA>           <NA>
4 65       Antibody        <NA>       Hormonal
5 88           <NA>        <NA>       Hormonal
6 78           <NA>       Other           <NA>
```

```
> dim(widechemo)
```

```
[1] 41  4
```

The number of rows of our reshaped wide data frame should be equal to `length(unique(x$id))`. *Note*, we needed to drop the `visitdt` column from consideration in the reshape. If we didn't, we would have gotten the following warning:

```
Warning message:
some constant variables (visitdt) are really varying in:
reshapeWide(data, idvar = idvar, timevar = timevar, varying = varying,
```

Let's work with another long format data frame that we wish to reshape to a wide format one. This example data frame has three columns: (1) subject ID; (2) week (1 - 10); and (3) hemoglobin (HGB) value. In this long format data frame, each subject has a HGB value for each week.

```
> xx <- data.frame(id = rep(1:100, times = 10), week = rep(1:10, each = 100),
+     hgb = sample(seq(4, 16, by = 0.1), size = 1000, replace = TRUE))
> dim(xx)
```

```
[1] 1000    3
```

```
> length(unique(xx$id))
```

```
[1] 100
```

```
> head(xx)
```

```
  id week  hgb
1  1    1  9.4
2  2    1  7.7
3  3    1 10.9
4  4    1 15.0
5  5    1  5.7
6  6    1  9.0
```

We want to reshape the data frame from long format to wide format such that there are 11 columns: subject ID, and one column for each week's visit.

```
> wide.xx <- reshape(xx, v.names = "hgb", idvar = "id", timevar = "week",
+     direction = "wide")
> head(wide.xx)
```

```
  id hgb.1 hgb.2 hgb.3 hgb.4 hgb.5 hgb.6 hgb.7 hgb.8 hgb.9 hgb.10
1  1   9.4  15.7  15.1  11.5  12.0   4.5  10.1   7.7   5.8    9.0
2  2   7.7  15.9   4.5   8.8  14.9   8.2  15.1   6.5   6.7    5.9
3  3  10.9   6.1   7.5  15.5   7.6  10.5   5.6  15.1   5.9    7.4
4  4  15.0  10.5  10.0  11.9  15.2  11.3   6.8   4.6  11.4   13.7
5  5   5.7   8.6  11.3   7.9   6.4   7.2   9.9   8.9  10.0    5.9
6  6   9.0  12.1   7.1   6.3  13.5   6.4   9.7   9.5   6.2   13.5
```

```
> dim(wide.xx)

[1] 100  11
```

Like before, the number of rows of our reshaped wide data frame should be equal to `length(unique(xx$id))`.

Let's take our `xx` long format data frame one step further by introducing some missing HGB values and missing records. For instance, let's drop 100 random records:

```
> subxx <- xx[sample(1:1000, size = 900, replace = FALSE), ]
```

And, in the remaining 900 records, let's replace 100 random values of HGB with `NA` (i.e., a missing value):

```
> subxx$hgb[sample(1:900, size = 100, replace = FALSE)] <- NA
```

Once again, let's reshape the long format data frame to a wide format data frame and see what's different.

```
> wide.subxx <- reshape(subxx, v.names = "hgb", idvar = "id", timevar = "week",
+     direction = "wide")
> head(wide.subxx)

    id hgb.8 hgb.4 hgb.6 hgb.1 hgb.3 hgb.10 hgb.2 hgb.9 hgb.5 hgb.7
703  3  15.1  15.5    NA  10.9   7.5    7.4    NA   5.9   7.6   5.6
399 99  10.2   4.7  14.5   5.8   8.0    9.6   4.6    NA  14.0   9.5
501  1   7.7  11.5   4.5   9.4  15.1     NA  15.7   5.8  12.0  10.1
37  37   4.8  10.3   9.1   6.3   6.9    4.9   9.0  10.1   9.0    NA
297 97   6.3    NA  10.9    NA   4.5     NA  13.7   6.1   7.6    NA
959 59    NA   9.9  10.2   5.4  10.2   12.9   7.7  10.6    NA    NA

> dim(wide.subxx)

[1] 100  11
```

Like twice before, the number of rows of our reshaped wide data frame should be equal to `length(unique(xx$id))`. But this time, you'll notice that the columns are no longer in order. You'll also notice that missing values are appropriately inserted in the reshaped data frame.

As a fourth example of reshaping a long format data frame to a wide format one, let's use a created data frame with two different variables (`x` and `y`) that have each been measured at two different time points (`Before` and `After`).

```
> df <- data.frame(id = rep(1:4, rep(2, 4)), visit = factor(rep(c("Before",
+     "After"), 4)), x = rnorm(4), y = runif(4))
> df

  id  visit          x         y
1  1 Before  1.0067290 0.2998300
2  1  After -1.6988995 0.8935240
3  2 Before -0.3840194 0.7626606
4  2  After -0.8087731 0.1040294
5  3 Before  1.0067290 0.2998300
6  3  After -1.6988995 0.8935240
7  4 Before -0.3840194 0.7626606
8  4  After -0.8087731 0.1040294

> reshape(df, timevar = "visit", idvar = "id", direction = "wide")
```

```
  id   x.Before  y.Before    x.After   y.After
1  1  1.0067290 0.2998300 -1.6988995 0.8935240
3  2 -0.3840194 0.7626606 -0.8087731 0.1040294
5  3  1.0067290 0.2998300 -1.6988995 0.8935240
7  4 -0.3840194 0.7626606 -0.8087731 0.1040294
```

Unlike the three previous examples, in this one we did not specify the `v.names=` argument – here, we did not need to since, after specifying the `timevar=` and `idvar=` arguments, only two columns remained to be considered. We can also see what happens when you reshape an "unbalanced" subset of a long format data frame.

```
> df2 <- df[1:7, ]
> df2

  id  visit          x         y
1  1 Before  1.0067290 0.2998300
2  1  After -1.6988995 0.8935240
3  2 Before -0.3840194 0.7626606
4  2  After -0.8087731 0.1040294
5  3 Before  1.0067290 0.2998300
6  3  After -1.6988995 0.8935240
7  4 Before -0.3840194 0.7626606

> reshape(df2, timevar = "visit", idvar = "id", direction = "wide")

  id   x.Before  y.Before    x.After   y.After
1  1  1.0067290 0.2998300 -1.6988995 0.8935240
3  2 -0.3840194 0.7626606 -0.8087731 0.1040294
5  3  1.0067290 0.2998300 -1.6988995 0.8935240
7  4 -0.3840194 0.7626606         NA        NA
```

As you noticed, `NA`s are used to 'fill in' the resulting wide format data frame.

Now let's discuss how to specify the arguments when `direction = "long"`. The `varying=` argument specifies the names of sets of variables in the current wide format that will correspond to single (time-varying) variables in the resulting long format. The `varying=` argument must be specified as a list of vectors. The `v.names=` argument can be optionally used to specify the name(s) of the variable(s) in the resulting long format that corresponds to the multiple variables in the original wide format. The `ids=` argument can be also be optionally used to specify the values to use for a newly created '`idvar`' variable in the resulting long format. And the `times=` argument can be also be optionally used to specify the values to use for a newly created '`timevar`' variable in the resulting long format.

With this in mind, let's work one `direction = "long"` example. For this example, we'll use the `MASS` package's (automatically installed) `immer` data frame, which is a *built-in* data set that we can access in R, and contains the yield data from a barley field trial. Specifically, the five varieties of barley were grown in six locations in each of 1931 and 1932. The `immer` data frame has 30 rows and 4 columns: (1) `Loc` the location; (2) `Var` the variety of barley (`manchuria`, `svansota`, `velvet`, `trebi`, and `peatland`); (3) `Y1` the yield in 1931; and (4) `Y2` the yield in 1932.

```
> library(MASS)
> data(immer)
> head(immer)

  Loc Var    Y1    Y2
1  UF   M  81.0  80.7
2  UF   S 105.4  82.3
```

```
3  UF   V 119.7  80.4
4  UF   T 109.7  87.2
5  UF   P  98.3  84.2
6   W   M 146.6 100.4
```

```
> dim(immer)
```

```
[1] 30  4
```

We want to reshape the wide format data frame to a long format one, such that the yield values from each year are placed in one column.

```
> immer.long <- reshape(immer, varying = list(c("Y1", "Y2")), direction = "long")
> head(immer.long)
```

```
    Loc Var time    Y1 id
1.1  UF   M    1  81.0  1
2.1  UF   S    1 105.4  2
3.1  UF   V    1 119.7  3
4.1  UF   T    1 109.7  4
5.1  UF   P    1  98.3  5
6.1   W   M    1 146.6  6
```

As you can see, just specifying the `varying=` argument doesn't generate the clearest labels, so let's specify some more arguments:

```
> immer.long <- reshape(immer, varying = list(c("Y1", "Y2")), timevar = "Year",
+     times = c(1931, 1932), v.names = "Yield", direction = "long")
> head(immer.long)
```

```
       Loc Var Year Yield id
1.1931  UF   M 1931  81.0  1
2.1931  UF   S 1931 105.4  2
3.1931  UF   V 1931 119.7  3
4.1931  UF   T 1931 109.7  4
5.1931  UF   P 1931  98.3  5
6.1931   W   M 1931 146.6  6
```

Now the resulting long data frame is much more descriptive.

In general, if a data frame (say `x`) resulted from a previous `reshape()` function invocation, then the operation can be reversed simply by using `reshape(x)`. In this case, the `direction=` argument is optional and the other arguments are stored as attributes on the data frame.

1.4 Data Export

When using R, you will often want to write specific data objects to a file. These objects may be formatted character strings that contain specific calculated results, or they may be whole matrices or data frames. There are many functions that can be used for data export, but in this section we will discuss the `cat()` and `write.table()` functions – see the 'Catalog' chapter for the other functions.

The most basic function for producing customized output is the `cat()` function, which concatenates and prints objects and character strings. And, when used in conjunction with other functions like the `round()`, and `format()` functions, it can print nicely formatted reports. The `cat()` function is also useful for producing output in user-defined functions. The syntax of the `cat()` function is

```
> args(cat)

function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
NULL
```

Like the `paste()` function, the `cat()` function converts it arguments (`...`) to character strings, concatenates them, separating them by the given `sep=` (separator) character string (by default, spaces), and then outputs them. For example,

```
> cat("The mean age of the subjects in our PBC data set is", round(mean(pbc$ageyrs),
+     1), "years.")

The mean age of the subjects in our PBC data set is 49.9 years.
```

Even though it is not shown here, by default, the `cat()` function does not go to a new line after being executed (`fill = FALSE`). In general, the `fill=` argument is a logical or (positive) numeric value controlling how the output is broken into successive lines. If `fill = FALSE` (default), only newlines created explicitly by `"\n"` are printed. Otherwise, the output is broken into lines based on the maximum printed line width that is set by the `width` option from the `options()` function (see the 'Customizing R' section of the second chapter for more details) if `fill = TRUE`, or the value of `fill=` if this is numeric. Based on this, the previous example can be modified to `cat("The mean age of the subjects in our PBC data set is", round(mean(pbc$ageyrs), 1), "years.", fill = TRUE)`, which is equivalent to `cat("The mean age of the subjects in our PBC data set is", round(mean(pbc$ageyrs), 1), "years.", "\n")`. In addition to newlines (`"\n"`), quotes and other special characters can be specified within character strings using *escape sequences*:

Escape Sequence	Character
`\'`	Single quote
`\"`	Double quote
`\n`	Newline
`\r`	Carriage return
`\t`	Tab character
`\b`	Backspace
`\a`	Bell
`\f`	Form feed
`\v`	Vertical tab
`\\`	Backslash itself

Here's another example that incorporates more escape sequences and the `round()` function,

```
> cat("Mean", "\t", "SD", "\t", "N", "\n", round(mean(pbc$ageyrs), 2),
+     "\t", round(sd(pbc$ageyrs), 2), "\t", sum(!is.na(pbc$ageyrs)), "\n")

Mean          SD          N
 49.92          11.89          100
```

Notice, the quotation marks are removed when the character strings are `cat()`ed. Also, as seen, by default, the result of a `cat()` function invocation is returned (printed) to the screen (`file = ""`), but it can also be written to an external file. Use the `file=` argument to specify a character string naming the file to print to. The file name can include a *path* like the file names specified in the `read.table()` function. Also, use `append = TRUE` to append the printed output to the file specified. By default, the `cat()` function overwrites the contents of the specified file (`append = FALSE`).

In addition to the `paste()` and `round()` functions demonstrated above, there are several other functions that can be incorporated into a `cat()` function expression to modify the output. Similar to the `round()` function, there are the `ceiling()`, `floor()`, `trunc()`, and `signif()` functions. Another useful function is the generic `format()` function, whose default method (`format.default()`) can format a numeric vector for 'pretty printing'. R does not always print output in the nicest, most consistent format. For example, by default, trailing zeros are removed from numeric values when they are printed, very large or small numbers are printed using scientific notation, and no thousand separators are used when large numbers are printed. For example, type the following numeric values at the command line and press return – `2.002`, `2.00`, `0.0000345`, and `34567901567`.

Luckily, the `format.default()` function has `nsmall=`, `scientific=`, and `big.mark=` arguments to help remedy these problems. Here are the same examples from above modified using the `format()` function.

```
> format(2, nsmall = 2)

[1] "2.00"

> format(3.45e-05, scientific = FALSE)

[1] "0.0000345"

> format(34567901567, big.mark = ",")

[1] "34,567,901,567"
```

There is also a `format.pval()` function that is also very helpful to use in conjunction with the `cat()` function.

A drawback to the `cat()` function is that it only prints vectors (anything else is coerced to a vector before outputted). Therefore, the `cat()` function is not very useful when you are trying to write non-vector output to a file, such as the output from a regression analysis. For this, I would suggest using the `sink()` function, which we discussed in the 'Diverting screen output to a file' section of the first document.

It is also useful to know how to write a data frame to a file in R, particularly in the instance where you have made several modifications to your data frame and would like to output it for use in some other program. Like reading in a data file, there are many functions in R that write a data frame to a file, and most are 'derivatives' of the `write.table()` function. The formal arguments of the `write.table()` function and their default values (if any) are:

```
> args(write.table)

function (x, file = "", append = FALSE, quote = TRUE, sep = " ",
    eol = "\n", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE,
    qmethod = c("escape", "double"))
NULL
```

By default, the specified data frame (`x=`) is written to the specified file (`file=`), as a space-delimited text file (`sep = " "`). In addition, both the column and row names are written (`col.names = TRUE` and `row.names = TRUE`, respectively), all character strings are quoted (`quote = TRUE`), and missing values are represented with `NA` (`na = "NA"`). For thoroughness, let's discuss some of the `write.table()` function's arguments and their useful modifications in more detail.

The `x=` argument is used to specify the object to be written, which is preferably a data frame or a matrix. If `x=` is not a data frame or a matrix, the `write.table()` function attempts to coerce the object to a data frame. As you can imagine, the `file=` argument is used to specify the name of the file (case sensitive, in quotes, and including its extension) to which the data should be written. Like in the `read.table()` function, the name of the file can be specified by only its name, or including its path – if no path is specified, the file is created in the current working directory. By default, the file the data frame is written to is overwritten if its already

exists (`append = FALSE`). Specify `append = TRUE` if the written data frame should be appended to the end of the specified file. By default, any character or factor columns of the data frame are surrounded by double quotes when the data frame is written to the specified file. In addition, if `quote = TRUE` column and/or row names (if printed) are also quoted. Specify `quote = FALSE` to suppress the quoting of such columns. As mentioned, by default, the data frame is written as a space-delimited text file (`sep = " "`). Like the `read.table()` function, the `sep=` argument specifies the *field separator character*, which is the character that will separate the values on each line of the written file. Use `sep = "\t"` to write tab-delimited files, and `sep = ","` for comma-separated value files ('.csv'). The `na=` argument is used to specify the character string to use to represent missing values in the written data. By default, missing values are represented with `NA` (`na = "NA"`). As mentioned, by default, the row names of the data frame are written to the specified file (`row.names = TRUE`). Specify `row.names = FALSE` to suppress this writing. The `row.names=` argument can also be specified as a character vector of row names to be written.

Lastly, the `qmethod=` determines how the `write.table()` function will deal with embedded double quote characters when quoting strings (e.g., `"This is a "double quote" within a quote"`). The `qmethod=` arguments must be specified as either `"escape"` (default) or `"double"`. Specifying `qmethod = "escape"` causes the quote character to be escaped in C style by a backslash (e.g., `"This is a \"double quote\" within a quote"`), while specifying `qmethod = "double"` causes the quote character to be doubled. When your data frame contains embedded double quotes, specifying `quote = FALSE` causes any embedded quotes to be written within an unquoted character column. As an example, let's build a data frame with a column that contains embedded quotes and then 'write' the data frame to a file specifying the `qmethod=` argument various ways. Note, when constructing a character vector in R, embedded double quotes (and other special characters within strings) are specified using the *escape sequences* `\"`. Also, specifying `file = ""` in the `write.table()` function invocation causes the data frame to be written to the screen.

```
> x <- data.frame(id = 1:2, comment = c("Double \"quote\" example 1",
+     "Another double \"quote\" example"))
> x

  id                         comment
1  1       Double "quote" example 1
2  2 Another double "quote" example

> write.table(x, file = "", qmethod = "escape")

"id" "comment"
"1" 1 "Double \"quote\" example 1"
"2" 2 "Another double \"quote\" example"

> write.table(x, file = "", qmethod = "double")

"id" "comment"
"1" 1 "Double ""quote"" example 1"
"2" 2 "Another double ""quote"" example"

> write.table(x, file = "", quote = FALSE)

id comment
1 1 Double "quote" example 1
2 2 Another double "quote" example
```

IMPORTING TEXT FILES INTO MICROSOFT EXCEL: It is quite easy to import data that is saved as a text file into Microsoft Excel. Specifically, you can import data into Microsoft Excel from most data sources by pointing to Import External Data on the Data menu, clicking Import Data, and then choosing the data you want to import in the Select Data Source dialog box – it is helpful if you choose All files (*.*) in the Files of type box before locating and double-clicking the text file you want to import in the Look in list. Microsoft Excel will then open the Text Import Wizard, which will allow you to specify how you want to divide the data into columns using various field separators. You can also reach the Text Import Wizard in a similar fashion by using the Open command in the File drop-menu.

Chapter 2

Working with R

Learning objective
To understand how to manage your workspace, customize your R sessions, use and avoid looping, evaluate expressions conditionally, and write your own functions.

OBJECT MANAGEMENT: During an R session, every object you assign a name is stored in what is known as your *workspace*. The `ls()` function can be used to display the names of all the variables currently assigned. For example, let's list the objects defined in our current workspace:

```
> ls()

 [1] "authors"    "books"      "char"       "df"         "df2"        "fac"
 [7] "fruit"      "immer"      "immer.long" "ourdf"      "ourlist"    "pbc"
[13] "subxx"      "widechemo"  "wide.subxx" "wide.xx"    "x"          "xx"
[19] "y"
```

As you can see, your workspace can quickly become cluttered, and it can become difficult to identify specific desired objects from among this clutter. Luckily, the `ls()` function has a `pattern=` argument that allows you to return the objects whose name matches a specific pattern. For example, we could list the objects that contain the letter 'x' using `ls(pattern = "x")`. The `pattern=` argument can be expanded to match based on regular expressions, which we discussed in the 'Vectors' manipulation section.

When you are assigning names to objects, it is important to know whether an object name is already being used (eg, is an existing function) before assigning it to a new object. The `exists()` function can be used to determine whether an object with a proposed name already exists – it searches for the name as a function or as the name of another assigned object. For example, `length` is the name of a function, but `lngth` has not been assigned yet.

```
> exists("length")

[1] TRUE

> exists("lngth")

[1] FALSE
```

The `rm()` function can be used to remove any unnecessary and/or unwanted objects by specifying the names of these objects. The name(s) may or may not be quoted, and multiple names can be specified if separated by commas. For example, to remove the object `x`, we would type `rm(x)` at the command line. To remove both the `x` and `y` objects, we would type `rm(x, y)`.

Multiple names can also be given in a vector form as a quoted string to the `list=` argument. Therefore, the previous example could be typed as: `rm(list=c("x", "y"))`. We can use this `list=` argument in conjunction with the `ls()` function to easily remove all of the objects in your workspace by typing `rm(list = ls())`. In the Windows version of R, the entire current workspace can also be cleared by selecting Remove all objects from the Misc drop-menu. We can also take this example one step further by incorporating the `setdiff()` function, which returns the difference between two sets of elements, to remove all objects except specified ones from your current workspace: `rm(list = setdiff(ls(), c(object1, object2)))`, where `object1` and `object2` are those objects you wish to keep.

→ *Practice Exercise:* Let's remove all the defined objects in our workspace except our `pbc` data frame.

```
> rm(list = setdiff(ls(), "pbc"))
> ls()

[1] "pbc"
```

There are also several ways to *save* desired objects. Recall, when you quit R, you are asked `Save your workspace image?` Accepting this offer will save all the objects in your current workspace to a *directory specific hidden file* named `.RData`[1]. If you saved your workspace before quitting and start R from the same directory at a later time, R loads this saved workspace and all of the objects are restored to the current workspace. However, I would highly recommend cleaning up your workspace using the `rm()` function before saving your workspace. The `save.image()` function can also be explicitly used to save the current workspace. By default, the objects are saved to the `.RData` hidden file, but a different file name can be specified using the `file=` argument. Unfortunately both of these methods are very inefficient and use a lot of memory to save the workspace.

An alternative is to use the `save()` function, which allows you to save specific objects and has an option to compress the file the objects are saved to. This is extremely useful when dealing with large objects that take significant execution time to create and/or normally take up scarce memory. Specify `compress = TRUE` to store the specified file very compactly – `save(list = c("object1", "object2"), file = "objects.rda", compress = TRUE)`, where `object1` and `object2` are those objects you wish to save, and `objects.rda` is the file you wish to compactly save them too. You then use `load("objects.rda")` to reload the saved objects at a later time (i.e., in another R session).

CUSTOMIZING R SESSIONS: The (possible) loading of objects from a saved image of the previous workspace from the hidden `.RData` file (if the workspace was saved) is just one step in the 'startup' procedure that R goes through when an R session is started. This startup procedure also sources (1) *environment files*, which contain lists of environment variables to be set, and (2) *profile files*, which contain R code used to load specific packages, to define important functions, and to define specific environment options, which affect the way in which R computes and displays its results. The nice thing is that we can modify the profile files sourced in this procedure in order to customize our R sessions. However, even though we can access the environment files, it is not recommended to modify these files.

After searching for and sourcing the environment files, R first searches for and sources a 'site-wide' startup profile file named `Rprofile.site`, which is located in the `R_HOME/etc` directory (if it exists). Recall, on a Linux, Unix, or Mac machine, `R_HOME` is `/usr/lib/R` or `/usr/local/lib/R`. On a Windows machine `R_HOME` is `C:\Program Files\R\R-version`, where `version` is the R version number (e.g., `2.5.1`). The `Rprofile.site` file is termed 'site-wide' because this file is sourced every time R is started, no matter what directory R is started in – so think of it as a 'global' file. In turn, the `Rprofile.site` files contains the expressions that you want to execute every time R is started anywhere on your machine. A second, 'personal', *hidden* file named `.Rprofile` can also be placed in any directory. In turn, R will source a hidden `.Rprofile` file if R is started in a directory that contains such a file. In contrast to the global `Rprofile.site` file, but

[1] By default, hidden files like `.RData` will not appear in a folder/directory. On a Unix/Linux machine, use `ls -a` at the shell command prompt to list all files, including hidden files. On a Windows machine, select Folder Options... from the View drop-menu. Then select Show all files from the View tab of the Folder Options dialog box.

like the hidden .RData file, the hidden .Rprofile file is 'directory specific' – it allows for different startup procedures in different working directories. You can make the hidden .Rprofile file slightly more global by saving it in your 'home' directory – if no hidden .Rprofile file is found in the current directory where R is started, then R looks for a hidden .Rprofile file in your home directory and uses that (if it exists). Remember, the .Rprofile file is hidden, so you will need to take similar measures as you did with the hidden .RData file to see it when you list a directory – see the previous footnote. Also, the Rprofile.site and hidden .Rprofile files often do not exist. Therefore, you can create these files using any text editor and then save them accordingly.

As mentioned, the Rprofile.site and hidden .Rprofile files contain R code used to load specific packages, to define important functions, and to define specific environment options, which affect the way in which R computes and displays its results. Specific add-on packages, like the Hmisc package, can be loaded whenever R is started by adding library() function expressions. You can also save R code to these files that defines functions that you have written yourself and that you wish to be able to use in your R sessions. In addition to any self-defined functions, you can also define two special functions – .First() and .Last() – in one or both of the profile files. The .First() function is automatically performed at the beginning of an R session and may be used to initialize the environment. Similarly, the .Last() function is executed at the very end of every session. The .First() function is where the specific environment options are defined using the options() function. The options() function contains a large number of arguments that control different aspects of the base, grDevices, stats, and utils packages. Like the par() function, modifying an argument of the options() function has a permanent effect, and we may modify specific arguments using a options(optionname = value, ...) construct. Some environment options you may consider modifying are:

- defaultPackages= specifies the packages, in addition to base, that are loaded when R is started – c("datasets", "utils", "grDevices", "graphics", "stats", "methods") by default.
- prompt= specifies the non-empty string to be used for R's command-line prompt – "> " by default and should usually end in a blank (" ").
- continue= specifies a non-empty string to be used for R's continuation prompt – "+ " by default.
- width= controls the maximum number of columns on a line used in printing data structures to the screen (and files) – 80 by default. This is useful when you re-size the window that R is running in; the Windows version of R automatically changes the value when the window is re-sized.
- digits= controls the number of digits to print when printing numeric values – 7 by default; valid values are from 1 to 22. Unfortunately, this is a suggestion only; not all functions will follow it. Also, there really is no way to specify that all numeric values should be *rounded* to so many decimal places by default. The alternative is using specific print() or format() functions.
- scipen=, specified using an integer, controls the penalty to be applied when deciding to print numeric values in scientific notation – 0 by default. Usually, decimals that are less than 0.001 are printed using scientific notation. Similarly, large numeric values greater than 1,000,000 are often printed with scientific notation. Specifying scipen= with positive integers will bias printing towards not using scientific notation (i.e., to scipen= decimal places). As with digits=, alternatively use specific print() or format() functions.
- stringsAsFactors= specifies the default setting for the stringsAsFactors= argument of the data.frame() and read.table() functions – recall, stringsAsFactors = TRUE by default, coercing all character columns to factors.

The options() function contains the full list, and can be returned (printed) by invoking the options() function with no arguments. You can use a getOption("optionname") construct to return (print) the value of a specific environment option. Environment options can also be set outside of the Rprofile.site and .Rprofile files.

Putting all of this information together, the following is an example of a Rprofile.site or .Rprofile file:

```
# Load the Hmisc package
library(Hmisc)

# Modify the command line and continuation prompt, and the width
options(prompt="$ ", continue="+\t", width = 100)

# Source a file that contains several self-defined functions
source(file.path(Sys.getenv("HOME"), "R", "mystuff.R"))

# Modify the q() function to automatically quit _without_ saving the workspace
#    (and without prompting)
q <- function(save = "no", status = 0, runLast = TRUE) {
   .Internal(quit(save, status, runLast))
}

.Last <- function() {
   # close all graphics window and file devices; a small safety measure
   graphics.off()
}
```

In the Windows version of R, you can also customize the way the R console 'looks and feels' using the `GUI preferences` selection under the `Edit` drop-menu.

CONDITIONAL EVALUATION: Up to this point, we have evaluated single expressions or multiple expressions grouped with the curly braces, `{ }`. However, we will often want to *conditionally* evaluates one or several expression. For example, perhaps we wish to calculate the mean of a continuous vector if the values are normally distributed, or the median if they are not. Conditional evaluation will also come in handy when you want to generalize your R code and perhaps incorporate it into a self-defined function. In R, an `if` statement allows us to evaluate expressions based on a condition, and takes one of two forms. The first is the 'if-then' form:

```
if(condition) {
   expression(s) if condition is TRUE
}
```

The `condition` is an expression that, when evaluated, returns a *single* `TRUE` or `FALSE` value – an error is returned if the `condition` does not evaluate to a logical value. If the `condition` evaluates to `TRUE`, then any `expression(s)` between the braces (`{` and `}`) is/are evaluated. The second form is an 'if-then-else' form:

```
if(condition) {
   expression(s) if condition is TRUE
else {
   expression(s) if condition is FALSE
}
```

In this form, the `condition` is evaluated, and any `expression(s)` between the *first set* of curly braces is/are evaluated if the `condition` evaluates to `TRUE`. If the `condition` evaluates to `FALSE`, then the `expression(s)` between the *second set* of curly braces is/are evaluated. The 'if-then-else' form of the `if` statement can also be *nested*.

```
if(condition1) {
   expression(s) if condition1 is TRUE
} else if(condition2) {
   expression(s) if condition 1 is FALSE but condition2 is TRUE
} else if(condition3) {
   expression(s) if both condition1 and 2 are FALSE but condition3 is TRUE
```

```
} else {
   expression(s) if condition1, 2, and 3 are FALSE
}
```

In any form, the `if` statement returns the value of the expression evaluated, or `NULL` if no expression was, which may happen if there is no `else`. When the `expression(s)` is/are not specified in a block involving braces (`{` and `}`), then the `else`, if present, *must* appear on the same line as the `if(condition)`. In general though, it is a good habit to *always* use braces to block the expressions with the appropriate part of the statement.

As mentioned, the `condition` of an `if` statement is expected to return a *single* `TRUE` or `FALSE` value when evaluated. If the `condition` returns a logical vector of more than one element, then a warning is given. For example,

```
> x <- c(2, 5, 7, NA, 10, NA, -1)
> if (x < 0) {
+     print("< 0")
+ } else {
+     print("> 0")
+ }
```

```
[1] "> 0"
```

```
Warning message:
the condition has length > 1 and only the first element will be used in: if (x < 0) {
```

In this example, the `condition` `x < 0` actually returns a logical vector the same length as `x`:

```
> x < 0
```

```
[1] FALSE FALSE FALSE    NA FALSE    NA  TRUE
```

When the `condition` evaluates to a logical vector of more than one element, as the warning suggests, only the first element of the evaluated `condition` (either `TRUE` or `FALSE`) is used – in this example, `FALSE`. Note, because it's a nonstandard topic name, the `if` statement must be quoted to access its help file – `help("if")`.

An alternative to the `if` statement is the `ifelse()` function, which is the vectorized version of the `if` statement. The `ifelse()` function has the form `ifelse(condition, expression1, expression2)` and returns a vector of the length of its longest argument, with elements `expression1[i]` if `condition` is `TRUE`, or `expression2[i]` otherwise. Here's an example:

```
> with(pbc, table(ifelse(ageyrs < 40, "< 40", ">= 40")))
```

```
 < 40 >= 40
   20    80
```

The `ifelse()` function can also be nested:

```
> with(pbc, table(ifelse(ageyrs < 40, "< 40", ifelse(ageyrs >= 40 & ageyrs <=
+     60, "40-60", "> 60"))))
```

```
 < 40 40-60  > 60
   20    59    21
```

The `ifelse()` function is very useful when you want to create new columns in a data frame that are derived from existing ones. Remember the 'rules' of how `TRUE` and `FALSE` values evaluate when combined using logic operators – see the 'Conditional expression' portion of the 'Vectors' manipulation section. Also remember the use of the `as.character()` function when conditionally evaluating factors with the `ifelse()` function — see the 'Coercing the Mode of a Vector' portion of the 'Vectors' manipulation section.

An alternative to the `if` statement and `ifelse()` function is the `switch()` function, whose syntax is

```
> args(switch)

function (EXPR, ...)
NULL
```

The `EXPR=` argument is an expression evaluating to a number or a character string and `...` is a 'list of alternatives' (not a formal list), which defines the output of the `switch()` function. The specific 'alternative' (i.e., element of the 'list') that is chosen is *conditionally* chosen based on the resulting value of `EXPR=`. Specifically, if `EXPR=` returns a number between 1 and length of the 'list' specified as `...` (i.e., the number of elements), then the corresponding element of the list is evaluated and the result is returned. If `EXPR=` returns a number that is less than 1 or is greater than the number of elements of the list, then `NULL` is returned. For example, let's conditionally evaluate a numeric vector depending on its length.

```
> x <- 10
> switch(length(x), x, median(x), mean(x))

[1] 10

> x <- c(5, 2)
> switch(length(x), x, median(x), mean(x))

[1] 3.5

> x <- c(3, 4, 7)
> switch(length(x), x, median(x), mean(x))

[1] 4.666667

> x <- sample(1:10, size = 5)
> switch(length(x), x, median(x), mean(x))

NULL
```

When the `EXPR=` argument evaluates to a character string, we need to name the elements of our 'list of alternatives', `....` In turn, the element of `...` with a name that exactly matches the resulting value of `EXPR=` is returned. If there is no match `NULL` is returned. For example,

```
> (x <- sample(c("horse", "cat", "dog"), size = 1))

[1] "horse"

> switch(x, horse = "Penny", cat = "Stripes", dog = "Spot")

[1] "Penny"
```

A common use of the `switch()` function is to branch according to the character value of one of the arguments to a function. For example,

```
> centre <- function(x, type) {
+     switch(type, mean = mean(x), median = median(x), trimmed = mean(x,
+         trim = 0.1))
+ }
> x <- rcauchy(10)
> centre(x, "mean")

[1] -4.81051

> centre(x, "median")
```

```
[1] 0.6287843

> centre(x, "trimmed")

[1] 0.4996638
```

REPETITIVE EVALUATION: In addition to conditionally evaluating expressions, we often want to repeatedly evaluate an expression or block expressions. This is often called '*looping*'. There are three statements that will perform looping: (1) the `for` statement; (2) the `while` statement; and (3) the `repeat` statement. However, the `while` and `repeat` statements are rarely used in R.

The standard `for` loop has the basic structure of

```
for(increment in sequence)
   expression(s)
}
```

The `increment` is a variable name – often the name of an indexer, such as `i`. The `sequence` can be any vector or list, and the `expression(s)` is/are evaluated for each `increment` (i.e., value) of `sequence`. When `sequence` is a vector, `increment` loops over each element of the vector (i.e., `sequence[increment]`). When `sequence` is a list, `increment` refers to each successive component in the list (i.e., `sequence[[increment]]`). A simple example is a countdown program:

```
> for (i in 5:1) {
+     print(i)
+ }

[1] 5
[1] 4
[1] 3
[1] 2
[1] 1

> i

[1] 1
```

In this example, `increment` is the variable `i`, and `sequence` is the numeric vector of the set of numbers 5 through 1. As seen, a `for` loop returns the value of the last expression evaluate (or `NULL` if none was), and sets `increment` to the last used element (component) of `sequence` (or to `NULL` if it was of length zero). Therefore, at the end of the `for` loop above, `i` now has the value `1`. `for` loops can also be nested – we merely have to use a different `increment` for each loop.

`for` loops are often used to evaluate an expression or block of expressions for specific columns or specific subsets of the rows of a data frame. For example,

```
> for (i in c("ageyrs", "fuyrs", "bili", "chol", "album")) {
+     cat("Mean of", i, "=", round(mean(pbc[[i]], na.rm = TRUE), 2), "\n")
+ }

Mean of ageyrs = 49.92
Mean of fuyrs = 5.12
Mean of bili = 3.11
Mean of chol = 381.62
Mean of album = 3.52
```

```
> for (i in c("ageyrs", "fuyrs", "bili", "chol", "album")) {
+     for (j in levels(pbc$sex)) {
+         cat("Mean of", i, "in", j, "patients =", round(mean(pbc[pbc$sex ==
+             j & !is.na(pbc$sex), i], na.rm = TRUE), 2), "\n")
+     }
+ }

Mean of ageyrs in Female patients = 49.07
Mean of ageyrs in Male patients = 55.16
Mean of fuyrs in Female patients = 5.13
Mean of fuyrs in Male patients = 5.04
Mean of bili in Female patients = 3
Mean of bili in Male patients = 3.79
Mean of chol in Female patients = 374.74
Mean of chol in Male patients = 417.27
Mean of album in Female patients = 3.5
Mean of album in Male patients = 3.63
```

In these examples we incremented across the elements of a character vector and took advantage of the double square brackets, `[[ ]]`, and `[ , ]` subsetting constructs to extract the rows and columns of our `pbc` data frame. As you can imagine, the `dfname$colname` construct does not work well with `for` loops.

IMPORTANT: Using a `for` loop is not always necessary in R. As we have already seen, many functions and operators are *vectorized*. In the 'Family of `apply()` functions' section , we will also encounter additional functions that perform *implicit* looping. In general, avoiding loops can make your R code more compact, easier to read, and often times more efficient in execution.

Let's get back to repetitive evaluation... Obviously, the `for` loop is great to use when you know ahead of time what sequence of values you want to loop over. However, sometimes you don't know this. In this case, you may want to repeat something as long as a condition is met (e.g., as long as a number is positive, or a number is larger than some tolerance). For this, use a `while` loop:

```
while(condition) {
   expression(s)
}
```

Like the `if` statement, the `condition` is an expression that, when evaluated, returns a *single* `TRUE` or `FALSE` value. If the `condition` evaluates to `TRUE`, then any `expression(s)` between the curly braces (`{` and `}`) is/are evaluated. This process continues until the `condition` evaluates to `FALSE`. Like the `for` loop, the `while` loop returns the value of the last evaluation of the `expression(s)`. For example,

```
> i <- 1
> while (i <= 5) {
+     cat("Iteration", i, "\n")
+     i <- i + 1
+ }

Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5

> i

[1] 6
```

If the `expression(s)` is/are never evaluated, then the `while()` function returns `NULL`. And like the `if` statement, a warning is returned if the `condition` returns a logical vector of more than one element (and only the first logical element of the evaluated `condition` is used).

Lastly, a `repeat` loop causes repeated evaluation of expressions until a break is specifically requested. This means that you need to be careful when using `repeat` because of the danger of an infinite loop. The syntax of the `repeat` loop is

```
repeat {
   expression(s)
}
```

Unlike `if`, `for`, and `while`, when using `repeat`, the `expression(s)` must be a block of code denoted with braces (`{` and `}`). This is because you need to both perform some computation and test whether or not to break from the loop, which usually requires at least two expressions.

The additional flow control statements `next` and `break` can be used to skip the next value in a loop or to break out of a loop, respectively. More specifically, `break` breaks out of a `for`, `while` or `repeat` loop, and control is transferred to the first statement outside the inner-most loop. `next` halts the processing of the current iteration and advances the looping index. Both `break` and `next` apply only to the innermost of nested loops.

The following example is a contrast and comparison of the `repeat` and `for` loops:

```
> i <- 0
> repeat {
+     if (i > 10)
+         break
+     if (i > 2 && i < 5) {
+         i <- i + 1
+         next
+     }
+     print(i)
+     i <- i + 1
+ }

[1] 0
[1] 1
[1] 2
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10

> for (i in 0:10) {
+     if (i > 2 && i < 5)
+         next
+     print(i)
+ }

[1] 0
[1] 1
[1] 2
[1] 5
```

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

Like the `if` statement, all three looping statements must be quoted to access their help files – e.g., `help("for")`.

'GROWING' DATA STRUCTURES IN A LOOP: We often generate data structures, such as vectors or data frames, in `for` loops. Unfortunately, if you're not careful, generating a data structure can be very memory intensive. Specifically, in each iteration of a `for` loop we often concatenate the next element of the vector onto the existing vector to generate the final vector. However, every time you do this, R implicitly copies the existing vector and then adds the additional element. Therefore, you are using `2n+1` the amount of memory, where `n` is the length of the vector during a specific iteration of the `for` loop, just to add a single element to the vector. Depending on the length `n`, this can be a huge amount of your memory. For example, we often do the following:

```
a <- NULL
for (i in 1:100) {
   a <- c(a, rnorm(1))
}
```

A much more efficient way of generating a vector in a `for` loop, is to define an 'empty' vector of the *final* length, if you know what this final length will be. We can then overwrite the value of each element with the correct value using the single square brackets, `[ ]`, subsetting construct. For example, suppose we want to generate a numeric vector of 100 elements. So, instead of the doing the above, we can do this:

```
a <- numeric(100)
for(i in 1:length(a)) {
   a[i] <- rnorm(1)
}
```

In contrast to 'growing' a vector, overwriting each element in a vector requires just the copying of the replacement elements. The `numeric()` function generates a numeric vector of specified length, where each element has a default value of 0. We could have also used the `character()` or `logical()` functions to generate a character/logical vector, respectively, of specified length. Each element of the character vector created with `character()` function has a default value of `""`, while each element of the logical vector created with `logical()` function has a default value of `FALSE`. For example,

```
> numeric(length = 10)

 [1] 0 0 0 0 0 0 0 0 0 0

> character(length = 10)

 [1] "" "" "" "" "" "" "" "" "" ""

> logical(length = 10)

 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

There are similar functions for complex, double, integer, raw, and real vectors, In addition, there is the `vector()` function, which produces a vector of the specified length *and mode*.

It is not necessary to know the final length of the vector you want to generate in the `for` loop. When assigning an element of a vector, if the number indexing the element to be assigned is greater than the length of the vector, the vector will be extended to that number of elements. In general, `NA`s are used to fill any gaps. For example,

```
> (x <- sample(1:10))

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

> x[15] <- 20
> x

 [1]  8  6 10  7  5  1  9  2  3  4 NA NA NA NA 20
```

Therefore, even if we don't know the final length of the vector we want to generate in a `for` loop, we can still define a vector of some length and then extend it. Similarly, we can always define a vector in a vector whose length may be too long and then shorten the vector once the `for` loop is complete.

We can use a similar process to efficiently generate a matrix (array) using the `[ , ]` (`[ , ,...]`) subsetting construct. For example,

```
> (mat <- matrix(NA, nrow = 2, ncol = 4))

     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA

> for (i in 1:nrow(mat)) {
+     for (j in 1:ncol(mat)) {
+         mat[i, j] <- rnorm(1)
+     }
+ }
> mat

           [,1]       [,2]       [,3]        [,4]
[1,] 0.08808937  1.4210358  1.3017982 -2.03791607
[2,] 1.64086197 -0.1099261 -0.8379594  0.07467112
```

Notice how R employed the recycling rule to generate our initial matrix of `NA`s. Unfortunately, we need to know the final dimensions of our desired matrix; we cannot expand the dimensions of matrix using the `[ , ]` subsetting construct.

Likewise, we need to know the final dimensions of a desired data frame. With a data frame, we can use the `numeric()`, `character()`, etc functions to define each column of the initial data frame. For example,

```
c <- data.frame(a = numeric(100), b = character(100))
```

Or, if you desired a data frame of a single mode (e.g., completely numeric), you can initialize the data frame like this:

```
b <- numeric(100)
attr(b, "dim") <- c(10,10)
c <- as.data.frame(b)
```

The `attr()` function is a way to define the attributes, like the dimensions, of a data structure. With either of these methods, you can then replace the various elements of the data frame using the `[[ ]]` or `[ , ]` subsetting constructs.

THE FAMILY OF `apply()` ***FUNCTIONS:*** As mentioned above, using a `for` loop is not always necessary in R. As we have already seen, many functions and operators are *vectorized*. In addition, there is a group of functions that allows us to perform *implicit* looping. This group of functions includes the `apply()` function and its 'relatives' `lapply()`, `sapply()`, and `tapply()`.

A common looping application is to apply a function to each element of a set of values or vectors and collect the results in a single structure. In R this is possible using the `lapply()` and `sapply()` functions. The `lapply()` function always returns a list (hence the 'l') whereas the `sapply()` function tries to simplify (hence the 's') the result to a vector or a matrix if possible. Both functions operate on the components of a list or the elements of a vector. In addition, both functions accept two main arguments: `X=` and `FUN=`, where `X=` specifies the data and `FUN=` specifies the function to be applied to each 'element' of `X=`. Additional arguments to the function specified using the `FUN=` argument can also be passed to the `lapply()`/`sapply()` functions via a `...` argument. So, to compute the range of the continuous variables of `pbc`, we can do the following:

```
> lapply(X = subset(pbc, select = c("fuyrs", "ageyrs", "bili", "chol",
+     "album")), FUN = range, na.rm = TRUE)

$fuyrs
[1]  0.5913758 12.1916496

$ageyrs
[1] 30.57358 78.43943

$bili
[1]  0.4 25.5

$chol
[1]  120 1128

$album
[1] 1.96 4.24

> sapply(X = subset(pbc, select = c("fuyrs", "ageyrs", "bili", "chol",
+     "album")), FUN = range, na.rm = TRUE)

          fuyrs   ageyrs bili chol album
[1,]  0.5913758 30.57358  0.4  120  1.96
[2,] 12.1916496 78.43943 25.5 1128  4.24
```

Notice how both functions attach meaningful names to the result, which is another good reason to prefer using these functions over using explicit loops. Also, notice how we passed `na.rm = TRUE` to the `range()` function.

The `lapply()` and `sapply()` functions can be used in conjunction with the `split()` function to generate results 'indexed' by a second vector. The `split()` function divides the values of a vector into the groups defined by a 'factor'. For example, we can calculate the mean `ageyrs` for each value of `drug`:

```
> with(pbc, split(ageyrs, drug))

$`D-penicillamine`
Age [years]
 [1] 42.50787 61.72758 33.63450 53.50856 32.61328 46.51608 67.31006 33.47570 76.70910
[10] 68.50924 41.15264 61.07050 48.85421 54.25599 35.15127 40.55305 42.96783 75.01164
[19] 69.94114 49.60438 41.60986 70.00411 62.62286 40.71732 61.29500 42.68583 59.76181
[28] 46.78987 44.82957 78.43943 31.44422 58.26420 51.48802 38.39836 50.10815 46.34908
[37] 67.57290 35.35113 35.53457 37.99863

$Placebo
Age [years]
 [1] 66.25873 59.95346 52.02464 41.38535 33.69473 49.13621 32.49281 55.83025 47.21697
```

```
[10] 34.03970 44.06297 42.63929 62.64476 34.98700 59.40862 42.74333 43.41410 35.49076
[19] 52.69268 56.77207 41.09240 49.76318 58.33539 41.37440 59.96988 74.52430 46.38193
[28] 45.08419 50.47228 56.15332 50.24504 31.38125 56.39151 30.57358 33.15264
```

```
> with(pbc, lapply(X = split(ageyrs, drug), FUN = mean, na.rm = TRUE))
```

```
$`D-penicillamine`
[1] 50.90849

$Placebo
[1] 47.76525
```

```
> with(pbc, sapply(X = split(ageyrs, drug), FUN = mean, na.rm = TRUE))
```

```
D-penicillamine         Placebo
       50.90849        47.76525
```

Another example would be to split a data frame with multiple records per subject on the subject IDs in order to calculate specific information on each subject (eg, the mean hemoglobin value for each subject).

A similar function to the `lapply()` and `sapply()` functions is the `appply()` functions, which allows you to apply a function to the rows or columns of a matrix (array) or data frame. Like the `lapply()`/`sapply()` functions, the `apply()` function wants a `X=` and `FUN=` argument, and it accepts additional argument to the function specified using the `FUN=` argument via a `...` argument. The `apply()` function also has a `MARGIN=` argument that is used to specify the dimension of `X=` the function `FUN=` will be applied over – `MARGIN = 1` indicates the rows, `MARGIN = 2` indicates the columns, and `MARGIN = c(1, 2)` indicates the rows and columns. For example, we could generate table of each of the factor variables in our `pbc` data set:

```
> apply(X = subset(pbc, select = c("drug", "sex", "ascites", "stage",
+     "censored")), MARGIN = 2, FUN = table)
```

```
$drug

D-penicillamine         Placebo
             40              35

$sex

Female   Male
    86     14

$ascites

 No Yes
 70   5

$stage

 1  2  3  4
 6 24 39 29

$censored

Censored     Dead
      68       32
```

With the `lapply()`, `sapply()`, and `apply()` functions, the function specified via the `FUN=` argument can return more than one value. In addition, the `FUN=` argument can be a self-defined function. For example, putting these two concepts together:

```
> with(pbc, sapply(X = split(ageyrs, drug), FUN = function(x) {
+     c(Mean = mean(x), SD = sd(x), Median = median(x), Min = min(x),
+         Max = max(x))
+ }))
```

```
       D-penicillamine  Placebo
Mean          50.90849 47.76525
SD            13.64798 10.99152
Median        47.82204 47.21697
Min           31.44422 30.57358
Max           78.43943 74.52430
```

In this case, the `sapply()` function returns a matrix. The `apply()` function would similarly return a matrix. Use the `lapply()` function if `FUN=` returns a list and make sure `X=` is specified as a list (i.e., e.g., the `subset()` function).

The last 'apply family' function to discuss is the `tapply()` function. The `tapply()` function is an alternative to using the `split()` function in conjunction with the `lapply()` or `sapply()` function. Like all the previous 'apply family' functions, the `tapply()` functions accepts an `X=` and a `FUN=` argument, and additional arguments to `FUN=` via a `...` argument. For the `tapply()` function, however, `X=` must specify a vector, and the `FUN=` argument must specify a function that returns only a single value. The `tapply()` function also accepts a `INDEX=` argument that allows you to specify *more than one* grouping 'factors'. In turn, the `tapply()` function creates tables (hence the 't') of the output of `FUN=` on subgroups of `X=` defined by `INDEX=`, which is specified using as *list*. If more than one 'factor' is specified, a cross-classified table is generated. In addition, if the 'factors' specified by `INDEX=` are not already defined as factors, they will be converted to factors internally. For example,

```
> with(pbc, tapply(X = ageyrs, INDEX = sex, FUN = median, na.rm = TRUE))
```

```
  Female     Male
47.10883 52.12320
```

```
> with(pbc, tapply(X = ageyrs, INDEX = list(sex, drug), FUN = median,
+     na.rm = TRUE))
```

```
       D-penicillamine  Placebo
Female        47.82204 44.57358
Male          51.46201 50.24504
```

An alternative to the `tapply()` function is the `aggregate()` function, which also splits the data (`x=`, one or more vectors) into specified subsets (using the `by=` argument, a *list* of grouping 'factors') and computes a value for each subset based on the value of the `FUN=` argument. Unfortunately, like the `tapply()` function, the `FUN=` argument is restricted to functions that return a single value. The advantage of the `aggregate()` function is that the result is reformatted into a *data frame*. This is often very useful, especially if you are grouping the output by subject ID – the output can be easily *merged* with the original data frame. For example, you may have two data frames – one representing the time in-varying information regarding each patient (e.g., age, race, gender, etc), the other representing time varying information (e.g., lab values for each visit). Using the `aggregate()` function, you can easily calculate the number of records in the time varying data frame for each patient and then merge this additional column of information (by patient ID) with the time in-varying data frame. The following is a dummy example:

```
> x <- data.frame(id = 1:10, age = sample(20:40, size = 10, replace = TRUE),
+     race = factor(sample(c("W", "B", "O"), size = 10, replace = TRUE)),
```

```
+     gender = factor(sample(c("M", "F"), size = 10, replace = TRUE)))
> y <- data.frame(id = sample(1:10, size = 100, replace = TRUE), lab = sample(100:500,
+     size = 100, replace = TRUE))
> (aggout <- with(y, aggregate(x = id, by = list(id), FUN = length)))

   Group.1  x
1        1 14
2        2  7
3        3  4
4        4 15
5        5  8
6        6  7
7        7 10
8        8 13
9        9 12
10      10 10

> x <- merge(x, aggout, by.x = "id", by.y = "Group.1", all = TRUE)
> x

   id age race gender  x
1   1  31    B      M 14
2   2  27    O      M  7
3   3  30    W      M  4
4   4  36    W      M 15
5   5  29    O      M  8
6   6  26    B      M  7
7   7  39    O      F 10
8   8  20    W      M 13
9   9  37    B      F 12
10 10  29    B      F 10
```

'APPLY' FUNCTIONS VERSUS LOOPING: In general, avoiding loops can make your R code more compact, easier to read, and often times more efficient in execution. We can test the performance of these functions using the `system.time()` function, which returns the CPU (and other) times that an expression used when it was being evaluated. For an example, let's compare various ways of subtracting the mean from each element in a 25000 x 4 matrix – the first way is using three `for` loops, while the second way uses the `apply()` function. This example comes from Kuhrnert & Venables' contributed document.

```
> mat <- matrix(rnorm(1e+05), ncol = 4)
> usingLoops <- function(mat) {
+     col.scale <- matrix(NA, nrow(mat), ncol(mat))
+     m <- NULL
+     for (j in 1:ncol(mat)) {
+         m[j] <- mean(mat[, j])
+     }
+     for (i in 1:nrow(mat)) {
+         for (j in 1:ncol(mat)) {
+             col.scale[i, j] <- mat[i, j] - m[j]
+         }
+     }
+     col.scale
+ }
> usingApply <- function(mat) {
+     apply(mat, 2, scale, scale = FALSE)
```

```
+ }
> system.time(usingLoops(mat))

   user  system elapsed
  0.804   0.000   0.803

> system.time(usingApply(mat))

   user  system elapsed
  0.012   0.004   0.016
```

Which approach would you use? The general rule of thumb is avoid loops, if it's possible and reasonable.

WRITING YOUR OWN FUNCTIONS: Even though R is a very powerful data analysis and manipulation tool, there will come a point at which you find yourself wanting to write your own functions. For example, you may want to change a built-in function to more appropriately fit your needs. Or perhaps, you would like to automate some of your code in order to be able to perform a specific task repeatedly and under various circumstances. The ability to write your own functions is one of the real advantages of R.

REVISITING FUNCTIONS: Recall, functions in R can do three things: (1) have values passed to them; (2) return a value; and/or (3) generate *side effects*, which are anything that is not the returning of a value (e.g., printing and plotting). Also, every function in R, whether intrinsic to the language or user-written, is defined using the same basic statement: `FUNname <- function( arglist ) { body }`, where `FUNname` is the name of the function; ***`arglist`*** is a *comma* separated list of zero or more arguments that can be passed to the function; and ***`body`*** contains the statements that perform the actions of the function. Except when the body of the function consists of a single expression, the body should always be enclosed between the curly braces (`{ }`). We can also define new binary operators, like `%in%`, in a similar fashion. To distinguish a binary operator from a function, the assigned name must be of the form `%anything%` and the name must be wrapped with quotation marks in the assignment – for example, `"%!%" <- function(X, y) {...}`.

Also keep in mind that an easy way to get started writing your own functions is to (copy/paste and) modify an existing function and assign it a new function name. In order to do this, we need to remember how to access the body of existing functions. As we know, typing the name of a function *without parentheses* at the command line and pressing return will return (print) the body of the function. Also recall generic functions, class specific methods, and'non-visible' functions – see the 'A Thought to End With: Object-Oriented Programming' section of the first document. Use the `getAnywhere()` function to print the body of a 'non-visible' function.

REVISITING FUNCTION ARGUMENTS: Recall, a function's arguments are treated like variables inside the function's body. Also recall, the arguments of a function are most often defined using an `ARGname` or an `ARGname = VALUE` construct. An `ARGname` argument is often the first argument in a function's argument list and often represents the main data object being passed to the function. You can also think of `ARGname` arguments as the ones that must always be specified in order for the function to evaluate. For example, in the `Hmisc` package's `smean.sd()` function, which returns (prints) the mean and standard deviation of a numeric vector, you must always specify the `x=` argument:

```
> smean.sd

function (x, na.rm = TRUE)
{
    if (na.rm)
        x <- x[!is.na(x)]
    n <- length(x)
    if (n == 0)
        return(c(Mean = NA, SD = NA))
    xbar <- sum(x)/n
```

```
    sd <- sqrt(sum((x - xbar)^2)/(n - 1))
    c(Mean = xbar, SD = sd)
}
<environment: namespace:Hmisc>
```

In contrast, `ARGname = VALUE` arguments are used to set a *default* value to an argument. In this case, the function will still evaluate if the `ARGname = VALUE` argument is not specified. However, we can modify the evaluation of the function if we modify `VALUE` to some other value. For example, in the `Hmisc` package's `smean.sd()` function, the `na.rm=` argument is set to a default value of `TRUE`, but we can easily modify this to `na.rm = FALSE` if we know the numeric vector we are evaluating contains no missing values.

As we have seen, the argument list can also have a special type of argument `...` ('dot dot dot'). This argument can hold a variable number of arguments and is mostly used for passing argument values to other functions that are invoked within the body of a function. For example, the argument lists of most high- and low-level plotting functions, like the `plot()` function, contain a `...` argument that is used to pass `par()` function arguments, like `pch=` or `lty=`, to the `par()` function that is called within the body of the plotting function. You can see where the arguments being 'absorbed' by the `...` argument are being passed internally by looking at the body of a function. For example, the default method of the generic `lines()` function:

```
> lines.default

function (x, y = NULL, type = "l", ...)
plot.xy(xy.coords(x, y), type = type, ...)
<environment: namespace:graphics>
```

Without a `...` argument, the argument list of our function would have to include each argument of each function called within the body of our function. Therefore, the `...` argument allows us to save a lot of coding and headache.

SAVING AND SOURCING FUNCTIONS: Obviously, to execute a function you wrote yourself, the function must be a defined object in your workspace (i.e., the name of the function is one of the names returned by the `ls()` function). To define the function in your workspace, you can always type the function assignment at the command line, or you can always copy and paste the function assignment from a text editor. If the function has been saved in a text file, it can be loaded with the `source()` function like other R code. Another possibility is to include the function assignment in your `RProfile.site` or `.RProfile` file. If you have several self-defined functions, you will want to consider whether you should save these in the same text file or save them to different text files.

LOCAL AND GLOBAL VARIABLES IN A FUNCTION: When writing a function, it is not necessary to declare the variables used within a function. When a function is executed, R uses a rule called *lexical scoping* to decide whether an object is local to the function's body or global (defined in your workspace). To understand this mechanism, let's consider a very simple function:

```
> printfun <- function() {
+     print(x)
+ }
> x <- 1
> printfun()

[1] 1
```

In this example, the name `x` is not used to create an object within the `printfun()` function. Therefore, R looks for an object called `x` from the (global) workspace. In fact, if `x` is not defined in the workspace, executing `printfun()` will result in an error – `Error in print(x) : object "x" not found`. In contrast, if `x` is assigned as the name of an object within our function, the value of `x` in the (global) workspace is not used. For example,

```
> x <- 1
> printfun <- function() {
+     x <- 2
+     print(x)
+ }
> printfun()

[1] 2

> x

[1] 1
```

In this example, the `printfun()` function used the object `x` that is defined within its *environment*, that is within its body. It is possible to create multiple nested environments (i.e., functions within functions) – the various environments enclose one another. In this case, the search for objects is made progressively from a given environment to the enclosing one, and so on, up to the global one.

It is also important to note that any 'ordinary' assignments (using `<-`) done within the a function are local and temporary and are lost after exit from the function. For example, in the following example, an error would be returned if we tried to return(print) the value of `y`.

```
> x <- 1
> printfun <- function() {
+     y <- 2
+     print(x + y)
+ }
```

RETURNING VALUES FROM A FUNCTION: As we know, functions are designed to return values. The value to be returned by a function may be given in an explicit call to the `return()` function or (more typically) is the value of the last expression in the body of the function. For example, the two following functions are equivalent:

```
oneFUN <- function(x, y = 5) {
   x + y
}
secondFUN <- function(x, y = 5) {
   return(x + y)
}
```

An object enclosed in the `return()` function will override the returning of the value of the last expression in the body of a function. For example,

```
> oneFUN <- function(x, y = 5) {
+     return(20)
+     x + y
+ }
> oneFUN(x = 5)

[1] 20
```

USEFUL FUNCTIONS FOR WRITING FUNCTIONS: There are many functions that are designed to be used only or primarily in the body of other functions. The `as._()` and `is._()` families of coercion and testing functions, respectively are very useful. As their names imply, they can be used to change the data type/structure of an object and to tell what kind of object the has been passed, respectively. The `missing()` function can be used to determine if the value of an argument of the form `ARGname` has been specified in the invocation of the function. The `missing()` function will return a single logical value of `TRUE` if the value of the argument was not specified. For example,

```
> examplefun <- function(x) {
+     if (missing(x)) {
+         return("x is missing")
+     }
+     else {
+         return("x is not missing")
+     }
+ }
> examplefun()

[1] "x is missing"

> examplefun(10)

[1] "x is not missing"
```

The `on.exit()` function records the expression given as its argument as needing to be executed when the current function exits (either naturally or as the result of an error). This is useful for resetting graphical parameters or performing other cleanup actions. For example,

```
parfun <- function(x, y) {
   oldpar <- par(no.readonly = TRUE)
   par(mar = c(2, 2, 2, 2) + 0.1, las = 1)
   plot(x, y)
   on.exit(oldpar)
}
```

The `stop()` and `warning()` handle errors and warnings, respectively. There are also several global options you can set using the `options()` function that pertain to errors and warnings: `'warn'`, `'warning.expression'`, and `'error'`

DEBUGGING FUNCTIONS: It is rare to write a function that works correctly the first time that it is tried. Often, all that is required to pinpoint the error is to add expressions to the body of the function that call the `print()` or `cat()` functions to print out partial results. When a function dies from an error, you can use the `traceback()` function to return (print) the sequence of calls that lead to the error, which is useful when an error occurs with an unidentifiable error message. Another option is the `debug()` function. To debug a problem function `fun()`, we execute the expression `debug(fun)`. We then execute the invocation of the function `fun()` that keeps causing an error – for example, `fun(x = 5, y = 2)`. Executing the problem expression invokes a debugger, which suspends normal execution of expressions and allows you to execute the body of the problem function one expression at a time. Before each expression in the body of the problem function is executed, the expression is printed and a special prompt is invoked. You can enter R expressions or special commands at the debugger prompt. The commands are `n` (or just return), which advances to the next expression in the body of the problem function; `c`, which continues to the end of the current context (e.g., to the end of the loop if within a loop or to the end of the function); `where`, which prints a stack trace of all the active function calls; and `Q`, which exits the debugger and returns to the command-line prompt. Anything else entered at the debug prompt is interpreted as an R expression to be evaluated in the calling environment. In particular typing an object name will cause the object to be printed, and `ls()` lists the objects in the calling frame. So, for example, if an object `x` is defined within the body of the problem function, typing `x` at the debug prompt will return (print) the value of `x`. If there is a local variable with the same name as one of the special commands listed above then its value van be accessed by using the `get()` function – e.g., `get("n")`. The debugger provides access only to interpreted expressions. If a function calls a foreign language (such as C) then no access to the statements in that language is provided. Debugging is turned off by a call to `undebug()` function with the function as an argument – e.g., `debug(fun)`.

SOME ADVICE: Like everything else in R, writing your own functions is a learning process, especially when it comes to writing functions that run faster and consume less memory. In addition to all the advice I have

tried to insert in this and the first document, the following is, as John Fox put it, some additional 'general, miscellaneous, and mostly unoriginal' advice about writing your own functions.

- *Work from the bottom up.* You will occasionally encounter a moderately large programming project. It is almost always helpful to break a large project into smaller parts, each of which can be programmed as an independent functions. In a truly complex project, these functions may be organized hierarchically, with some calling others. If some of these small functions are of more general utility, then you can maintain them as independent functions and reuse them.

- *Test your program.* Before worrying about speed, memory usage, elegance, and so on, make sure that your function provides the right answer. Developing any function is an iterative process of refinement, and getting a function to work correctly is the key first step. In checking out your function, try to anticipate all of the circumstances that the function will encounter and test each of them. Furthermore, in 'quick-and-dirty' programming, the time that you spend writing and debugging your function will probably be vastly greater than the time the function spends executing. Remember the programmer's adage: 'Make it right before you make it faster.'

- *Document your functions.* The best way to document your functions is to write them in a transparent and readable style – use descriptive function and argument names, even if this means that they are longer than what you may prefer; name the elements of the various data structures that are created in the body a function (i.e., `x[, "dead"]` is more meaningful than `x[ , 2]`); avoid clever but cryptic tricks in your code (trust me, you'll spend a fair amount of time trying to figure out the trick every time you read your code); break long expressions over multiple lines; indent lines (for example, in loops) to reveal the structure; always use parentheses to make groupings explicit; always use curly braces, `{` and `}`, in the body of your functions, including with `for`, `if`, `else`, and other statements; and always use `TRUE` and `FALSE` instead of `T` and `F`. In addition, if at all possible, give parameters sensible defaults. You can also add a few comments to the beginning of a function to explain what the function does and what its arguments mean. The key is that you want to understand your own functions when you return to them a month or a year later.

Chapter 3

Catalog of Functions, Operators, & Constants

This chapter is intended to act as a reference regarding the functions, operators, and constants we have discussed in both documents. This catalog also includes additional functions that we did not cover, but I feel would be very useful to be aware of. ***NOTE:*** This catalog does not include any graphics functions; I have compiled the graphical material into its own reference, which you can find in the next chapter. The functions, operators, and constants given in this chapter come from the `base`, `foreign`, `Hmisc`, `stats`, and `utils` packages; each item is labeled with its corresponding package. ***ALSO NOTE:*** This chapter does not include all of the functions available from these packages, just the ones I have found I use the most or are most likely to use. The functions, operators, and constants in the chapter have also been grouped by topics, such as data creation or data manipulation. It is also important to be aware of the set of *manuals* available on the R website (`http://www.r-project.org/`):

- **R Installation and Administration:** Comprehensive overview of how to install R and its packages under different operating systems.
- **An Introduction to R:** Provides an introduction to the language.
- **R Data Import/Export:** Describes import and export facilities.
- **Writing R Extensions:** Describes how you can create your own packages.
- **The R Reference Index:** Contains printable versions of all the R help files for standard and recommended packages.

The manuals can be accessed by clicking on the Manuals link listed under Documentation.

R Session

Entry	Package	Description
`Startup`	`base`	Describes the initialization at the start of an R session (help topic only)
`R.home()`	`base`	Returns the `R_HOME` directory
`options()`	`base`	Returns all options settings or sets specific options settings
`getOption()`	`base`	Returns the current value of specific options settings
`getwd()`	`base`	Returns path of current working directory
`setwd()`	`base`	Sets path to desired working directory
`list.files()`	`base`	Lists the files in the current working directory
`date()`	`base`	Returns system's current date and time
`Sys.Date()`	`base`	Returns system's current date
`Sys.time()`	`base`	Returns system's current time

`Sys.timezone()`	`base`	Returns system's current time zone
`R.Version()`	`base`	Returns detailed information about the version of R running
`sessionInfo()`	`utils`	Returns version information about R and loaded (attached) packages
`history()`	`utils`	Displays the command history of the current R session
`timestamp()`	`utils`	Writes a timestamp (or other message) into the command history of the current session and echos it to the console
`savehistory()`	`utils`	Saves the command history of the current R session to a specified file
`loadhistory()`	`utils`	Loads the saved command history from a previous R session (from a specified file)
`source()`	`base`	Reads and executes R code from a file
`system()`	`base`	Invokes a system command
`q()`	`base`	Terminates an R session

Finding Help

Entry	Package	Description
`args()`	`base`	Returns the argument list of the specified function
`argsAnywhere()`	`utils`	Returns the argument list of the specified 'non-visible' function
`methods()`	`utils`	Returns the class specific methods of a generic function
`getAnywhere()`	`utils`	Returns the body of the specified 'non-visible' function
`help()`	`utils`	Provides access to the help file of the specified topic; alternatively `?`
`help.search()`	`utils`	Searches the help files for the specified character string
`RSiteSearch()`	`utils`	Searches for key words or phrases in the R-help mailing list archives or documentation
`help.start()`	`utils`	Starts the hypertext (currently HTML) version of R's online help files
`apropos()`	`utils`	Returns a character vector giving the names of all objects (functions and assigned objects) in the search list or that match a specified character string
`find()`	`utils`	Similar to `apropos()`, but allows you to specify the mode of the objects to return
`example()`	`utils`	Executes the 'Example' section from the specific help file
`demo()`	`utils`	Executes some demonstration R code for a specific topic, or lists the available demos
`vignette()`	`utils`	Similar to `demo()`
`RShowDoc()`	`utils`	Shows R manuals or other documentation
`Syntax`	`base`	Describes the operator syntax and precedence (help topic only)
`Paren`	`base`	Describe how R handles parentheses and braces (help topic only)
`Quotes`	`base`	Describes the various uses of quoting in R (help topic only)

Packages

Entry	Package	Description
`R.home()`	`base`	Returns the `R_HOME` directory
`installed.packages()`	`utils`	Finds (or retrieves) details of all packages installed in the specified libraries
`install.packages()`	`utils`	Installs packages from CRAN or a downloaded package source/binary file

update.packages()	utils	Compares current versions of installed packages, asks if you would like to update each package that has a newer version available, and updates the packages you specify to update
remove.packages()	utils	Removes specified installed packages
library()	base	Loads specified package
search()	base	Returns a character vector of the currently loaded packages
searchpaths()	base	Returns a character vector of the currently loaded packages with their directory paths
detach()	base	Un-loads the specified package

Data Creation

Entry	Package	Description
Assignment	base	<-, ->, << −, − >>, and the assign() function
NA	base	'Not available'; representation of missing values
NULL	base	The null object
LETTERS, letters	base	Generates a character vector of the specified subset of the 26 upper- and lower-case letters of the Roman alphabet
month.abb, month.name	base	Generates a character vector of the specified subset of three-letter abbreviations or full names of the months of the year
c()	base	Generates a vector or list by concatenating the specified values
Cs()	Hmisc	Generates a character vector by concatenating the specified values
character()	base	Generates a character vector of the specified number of elements
complex()	base	Generates a complex vector of the specified number of elements
double(), single()	base	Generates a double-/single-precision vector of the specified number of elements
integer()	base	Generates an integer vector of the specified number of elements
logical()	base	Generates a logical vector of the specified number of elements
numeric()	base	Generates a numeric vector of the specified number of elements
raw()	base	Generates a raw vector of the specified number of elements
real()	base	Generates a real vector of the specified number of elements
vector()	base	Generates a vector of the specified number of elements and mode
paste()	base	Generates a character vector concatenating the specified vectors
sample()	base	Generates a vector by taking a random sample of the specified elements (used in conjunction with the set.seed() function)
rep()	base	Generates a vector by replicating the specified elements
score.binary()	Hmisc	Generates a vector from a series of logical conditions
seq(), : operator	base	Generates a regular sequence
sequence()	base	Generates a vector of sequences
combn()	utils	Generates a vector of all the combinations of 'n' elements, taken 'm' at a time
cut()	base	Generates a factor by dividing the range of a numeric vector into intervals and coding the values of the numeric vector according to which interval they fall
factor()	base	Encodes a vector as a factor
gl()	base	Generates a factor by specifying the pattern of its levels
interaction()	base	Generates a factor which represents the interaction of the specified levels
matrix()	base	Creates a matrix
array()	base	Creates an array
rbind(), cbind()	base	Combines vector, matrices, or arrays by rows or columns
data.frame()	base	Creates a data frame

expand.grid()	base	Creates a data frame from all the combinations of the specified vectors or factors
list()	base	Creates a list

Data Import

Entry	Package	Description
data(), getHdata()	utils, Hmisc	Loads the specified built-in data set, or lists the available data sets
read.table()	utils	Reads in a white space delimited file
read.delim()	utils	Reads in a tab-delimited file
read.csv(), csv.get()	utils, Hmisc	Reads in a comma-delimited file
read.dta(), stata.get()	foreign, Hmisc	Reads in a Stata file
read.spss(), spss.get()	foreign, Hmisc	Reads in a SPSS file
read.xport(), read.ssd(), sasxport.get()	foreign, Hmisc	Read in a SAS Transport file
mdb.get()	Hmisc	Reads in tables from a Microsoft Access Database file
read.epiinfo()	foreign	Reads in an Epi Info data file
read.ftable()	stats	Reads in a flat contingency table
read.fwf()	utils	Reads in a fixed-width format file
read.mtp()	foreign	Reads in a Minitab Portable Worksheet file
read.octave()	foreign	Reads in an Octave text data file
read.systat()	foreign	Reads in a Systat file
scan()	base	Reads data into a vector or list from the command line or file

Data Attributes

Entry	Package	Description
attr()	base	Returns or sets the specific attributes of an object
attributes()	base	Returns the attributes of an object
class(), data.class()	base	Returns (or sets) the class of an object
contents()	Hmisc	Returns the 'metadata' of the columns of a data frame, which includes the names, labels (if any), units (if any), number of factor levels (if any), factor levels, class, storage mode, and number of NAs
dim(), nrow(), ncol()	base	Returns (or sets) the dimensions (number of rows and columns) of an object
dimnames(), rownames(), colnames()	base	Returns or sets the dimension names (row and column names) of an object
row.names()	base	Returns or sets the row names for data frames
is._()	base	Tests whether an object is something – execute apropos("^is\\.") to see all the available functions, including is.na(), is.nan(), and is.null()
label()	Hmisc	Returns or sets the label of an object
length()	base	Returns or sets the length of vectors (including factors) and lists
levels(), nlevels()	base	Returns or sets the levels of a factor; returns the number of levels of a factor

`mode()`, `storage.mode()`	`base`	Returns or sets the mode and storage mode of an object
`names()`	`base`	Returns or sets the names of the elements/components of an object
`str()`	`utils`	Compactly displays the structure of an object
`typeof()`	`base`	Returns the type of an object
`units()`	`Hmisc`	Returns or sets the units of an object
`upData()`	`Hmisc`	Updates a dataframe and the attributes of its columns

Data Manipulation: Math

Entry	Package	Description
Numeric constants	`base`	`Inf`, `NaN`, and `pi`
Arithmetic operators	`base`	`+`, `-`, `*`, `/`, `^` (exponentiation), `%%` (modulo), and `%/%` (integer division)
`diff()`	`base`	Calculates the lagged and iterated differences between the elements of a numeric vector
`sum()`	`base`	Sums the elements of a vector
`prod()`	`base`	Calculates the product of all the elements of a numeric vector
`cumsum()`	`base`	Calculates the cumulative sums of the elements of a numeric vector
`cumprod()`	`base`	Calculates the cumulative products of the elements of a numeric vector
`cummax()`	`base`	Calculates the cumulative maximas of the elements of a numeric vector
`cummin()`	`base`	Calculates the cumulative minimas of the elements of a numeric vector
`abs()`	`base`	Calculates the absolute values of the elements of a numeric vector
`sqrt()`	`base`	Calculates the square roots of the elements of a numeric vector
Trigonometric & Hyperbolic	`base`	`cos()`, `sin()`, `tan()`, `acos()` ('a' = arc), `asin()`, `atan()`, `atan2()`, `cosh()` ('h' = hyperbolic), `sinh()`, `tanh()`, `acosh()`, `asinh()`, and `atanh()`
Logarithms & Exponentials	`base`	`log()`, `logb()`, `log1p()`, `log10()`, `log2()`, `exp()`, and `expm1()`
Combinatorics	`base`	`choose()`, `lchoose()`, `factorial()`, and `lfactorial()`
Matrix operators	`base`	`%*%` (matrix multiplication), `%o%` (outer product), and `%x%` (Kroenecker product)
Other matrix & array related	`base`	`colSums()`, `rowSums()`, `rowsum()`, `colMeans()`, `rowMeans()`, `crossprod()`, `tcrossprod()`, `det()`, `eigen()`, `diag()`, `lower.tri()`, `upper.tri()`, `qr()`, `svd()`, `scale()`, `t()`, `svd()`, `aperm()`, and `sweep()`

Data Manipulation: Descriptive Statistics

Entry	Package	Description
`cor()`	`stats`	Calculates the correlation between two numeric vectors or matrices
`cov()`	`stats`	Calculates the covariance between two numeric vectors or matrices
`var()`	`stats`	Calculates the variance between two numeric vectors or matrices
`cov2cor()`	`stats`	Scales a covariance matrix into the corresponding correlation matrix
`density()`	`stats`	Calculates the kernel density estimates of a numeric vector
`ecdf()`	`stats`	Calculates the empirical cumulative distribution function (ECDF) of a numeric vector

min(), max(), range()	base	Calculates the minimum, maximum, and both values of a numeric vector
pmin(), pmax()	base	Calculates the parallel minima and maxima of two or more numeric vectors or matrices
quantile()	stats	Calculates various sample quantiles of a numeric vector
IQR()	stats	Calculates the inter-quartile range of a numeric vector
fivenum()	stats	Calculates Tukey's five-number summary (minimum, lower-hinge, median, upper-hinge, and maximum) of a numeric vector
median()	stats	Calculates the median of a numeric vector
mean()	base	Calculates the arithmetic mean of a numeric vector
weighted.mean()	stats	Calculates the weighted mean of a numeric vector
mad()	stats	Calculates the mean absolute difference of a numeric vector
sd()	stats	Calculates the standard deviation of a numeric vector
rank()	base	Calculates the sample ranks of the values of a vector
smean.sd()	Hmisc	Calculates the mean and standard deviation of a numeric vector
smean.cl.normal()	Hmisc	Calculates the sample mean and lower and upper Gaussian confidence limits of a numeric vector, based on the t-distribution
smean.sdl()	Hmisc	Calculates the mean plus or minus a constant times the standard deviation
smean.cl.boot()	Hmisc	A very fast implementation of the basic nonparametric bootstrap for obtaining confidence limits for the population mean without assuming normality
smedian.hilow()	Hmisc	Calculates the sample median and a selected pair of outer quantiles having equal tail areas
wtd.mean	Hmisc	Calculates the weighted mean of a numeric vector
wtd.var	Hmisc	Calculates the weighted variance of a numeric vector
wtd.quantile	Hmisc	Calculates the weighted quantiles of a numeric vector
wtd.Ecdf	Hmisc	Calculates the weighted ECDF of a numeric vector
wtd.rank	Hmisc	Calculates the weighted ranks of a numeric vector, using mid-ranks for ties
describe()	Hmisc	Provides a concise statistical description of a vector, matrix, or data frame
bystats(), summary.formula()	Hmisc	Generates descriptive statistics by categories

Data Manipulation: Dates

Entry	Package	Description
as.Date()	base	Converts a character vector to a Date class vector
strptime()	base	Converts a character vector to a POSIXlt class vector
as.POSIXct()	base	Converts a POSIXlt class vector to a POSIXct class vector
cut()	base	Converts a datetime class object to a factor (see .Date and .POSIXt methods)
rep()	base	Replicates the elements of datetime class vectors (see .Date and .POSIXt methods)
seq()	base	Generates a regular sequence of dates (see .Date and .POSIXt methods)
format()	base	Formats how datetime class vectors are printed (see .Date and .POSIXt methods)
round(), trunc()	base	Rounds/truncates datetime class vectors (see .Date and .POSIXt methods)
diff(), difftime()	base	Calculates the lagged and iterated differences between the elements of a datetime class vector (see .Date and .POSIXt methods)

Data Manipulation: Tables

Entry	Package	Description
table()	base	Creates a one-, two, or n-way table
xtabs()	stats	Alternative to the table() function (has formula interface)
ftable()	stats	Creates a 'flat' 3-way or greater table
prop.table()	base	Expresses table entries as proportions of margins
margin.table()	base	Computes the sum of the table entries for a given margin
addmargins()	stats	Expands a n-way table to include margin totals

Other Data Manipulation

Entry	Package	Description
with()	base	Evaluates an expression in a data environment
format()	base	Format an R object for pretty printing (generic)
as._()	base	Coerces an object (generic; execute apropos("^as\\.") for complete list)
I()	base	Inhibits coercion of objects
unlist()	base	'Flattens' lists
levels(), relevel(), reorder()	base, stats	Modifies the levels of a defined factor
ifelse()	base	Conditionally generates a vector from existing vectors
transform()	base	Transforms an object, such as a data frame
mChoice()	Hmisc	Generates an object representing non-mutually exclusive events
union(), intersect(), setdiff(), setequal()	base	Performs set operations
order(), sort(), rev()	base	Sorts.orders the elements of a vector
merge()	base	Merges two data frames by common columns or row names
reshape()	stats	Reshapes a data frame from 'long' to 'wide' and vice versa
round(), signif(), ceiling(), floor(), trunc()	base	Rounds numeric values in various ways
aggregate()	stats	Splits the data into subsets, computes the value of a function for each, and returns the result in a data frame
by()	base	Applies a function to a data frame split by factors
apply(), lapply(), sapply(), tapply()	base	Applies a function over the elements of an object
split()	base	Divides a vector into groups based on a factor
complete.case()	stats	Returns a logical vector indicating which cases (rows) of an object are complete (i.e., have no missing values)
unique()	base	Returns the vector or data frame with duplicate elements removed
duplicated()	base	Determines which elements of a vector or data frame are duplicates of elements with smaller subscripts and returns a logical vector indicating which element (rows) are duplicates
head()	utils	Returns the first/last parts of a vector, matrix, or data frame
subset()	base	Returns subsets of vectors or data frames which meet conditions
Extract	base	Subsetting operators: [ ], [[ ]], [,], and $
Logic	base	! (negation), & and && (AND; intersection), | and || (OR; union), and the xor() and isTRUE() functions
Matching	–	%in% (base), %nin% (Hmisc), and is.element() (base)

`all()`, `any()`	`base`	Given a set of logical vectors, are all/any (at least one) of the values `TRUE`?
`all.equal()`, `identical()`	`base`	Tests if two objects are nearly/exactly equal
`which()`	`base`	Returns the indices of a logical object that are `TRUE`
`which.max()`, `which.min()`	`base`	Returns the indices of a numeric vector of the (first) occurrence of the maximum or minimum
`tolower()`, `toupper()`	`base`	Character translation from upper to lower case or vice versa
`strsplit()`	`base`	Splits the elements of a character vector into substrings according to the presence of a substring within them
`grep()`, `sub()`, `gsub()`	`base`	Pattern matching and replacement (based on regular expressions)

Data Export

Entry	Package	Description
`cat()`	`base`	Outputs the specified information as a single concatenated character vector either to the screen or to a specified file
`print()`	`base`	Prints (returns) the specified object to the screen
`sink()`	`base`	Diverts R output from the screen to a specified file
`write()`	`base`	Writes the specified data (usually a matrix) to the specified file
`write.table()`	`utils`	Writes the specified data frame to the specified file

Object Management

Entry	Package	Description
`exists()`	`base`	Returns `TRUE/FALSE` depending on whether object is already defined
`ls()`	`base`	Lists the currently assigned objects
`rm()`	`base`	Removes specified assigned objects
`save()`	`base`	Saves specified assigned objects to specified file
`save.image()`	`base`	Shortcut for saving your current workspace to the hidden `.RData` file
`load`	`base`	Loads saved objects from specified file
`attach()`	`base`	Attaches the specified list or data frame to the 'search path' (output of the `search()` function)
`detach()`	`base`	Removes the specified list or data frame to the 'search path'
`search()`	`base`	Returns a character vector of the currently loaded packages and any attached lists or data frames
`conflicts()`	`base`	Searches for *masked* objects (objects that exist with the same name in two or places on the 'search path')
`mem.limits`	`base`	Controls the memory available for R
`Memory-limits`	`base`	Describes the memory limits of R (must specify in quotes to access help file – `?"Memory-limits"`)
`object.size()`	`utils`	Provides an estimate of the memory that is being used to store the specified R object

Writing Functions

Entry	Package	Description
`Assignment`	`base`	`<-`, `->`, $<< -$, $- >>$, and the `assign()` function
`alarm()`	`utils`	Gives an audible or visual signal to the user
`missing()`	`base`	Tests whether a value was specified for a specific argument to a function
`match.arg()`	`base`	Matches the argument specified against a table of candidate values
`match.call()` `do.call()`	`base`	Constructs and executes a function call from a name or a function and a list of arguments to be passed to it
Repetitive & Conditional evaluation	`base`	`if`, `else`, `for`, `while`, `repeat`, `break`, `next` (must specify in quotes to access help file – `?"for"`), and the `switch()` function
`stop()`	`base`	Stops the execution of the current expression and executes an error action
`warning()`	`base`	Generates a warning message that corresponds to its argument(s)
`require()`	`base`	Loads the specified package if not already done so
`tempfile()`	`base`	Creates names for temporary files
`return()`	`base`	Returns a value from a function
`invisible()`	`base`	Returns a (temporarily) invisible copy of an object
`on.exit()`	`base`	Records the expression given as its argument as needing to be executed when the current function exits (either naturally or as the result of an error)
`traceback()`	`base`	Prints the sequence of calls that lead to the error of a function
`debug()`	`base`	Debugs a function
`Rprof()`	`utils`	Profiles the execution of R expressions

Chapter 4

R Graphics Reference

HIGH-LEVEL PLOTTING FUNCTIONS: Recall, a traditional graphics plot is created by first calling a *high-level* plotting function that creates a 'complete' plot. This means that with high-level plotting functions, axes, labels and titles are automatically generated, where appropriate, and unless you request otherwise. In addition, high-level plotting functions always start a new plot, erasing the current plot if necessary. The following table lists the high-level plotting functions available in the `graphics` and `grDevices` packages. The list also contains some additional high-level plotting functions from the `stats` and `Hmisc` packages – denoted with their corresponding package in parentheses. Examples of specific high-level plotting functions (denoted with an asterisk, *) are shown in Figures 4.1 to 4.5.

Function	Description
`curve()`	Generates a curve corresponding to the given function or expression. Alternative is the `plot.curve()` method of the generic `plot()` function.
`hist()`	Generates a histogram of the given data points, including POSIXt and Date class data. Alternatives include the `Hmisc` package's `hist.data.frame()` and `histbackback()` functions.
`boxplot()`	Used to plot the 'five-number' summary of a continuous variable and can generate side-by-side boxplots. Can be used in conjunction with the `bxp()` function. Also see the `bpplt()` (`Hmisc`) function. Uses the `boxplot.stats()` function to gather the statistics necessary for producing the boxplot.
`stripchart()`	Generates '1D scatter plots' or dotplots of continuous data. Alternative is the `stem()` function.
`interaction.plot()` (`stats`)	Plots the mean (or other summary statistic) of a continuous (response) variable for a two-way combination of (two) categorical variables, thereby illustrating possible interactions.
`barplot()`	Graphically displays the relative frequencies or proportions of the 'levels' of a categorical variable and can generate stacked or side-by-side barplots representing the 'relationship' between two categorical variables (i.e., a two-way contingency table).
`pie()`	Generates a pie chart, displaying the relative frequencies or proportions of the 'levels' of a categorical variable.
`dotchart()`	Generates a dotplot, an alternative to barplots.
`mosaicplot()`*	Generates a mosaic-plot of a one-way or greater contingency table.
`spineplot()`*	Generates spine plots and spinograms, which are special cases of mosaic plots. Also seen as generalizations of stacked (or highlighted) bar plots.
`fourfoldplot()`*	Creates a 'four-fold display' for the special case of two dichotomous categorical variables grouped by a third categorical variable with at least two levels (i.e., data from a 2 by 2 by k three-way table).

`assocplot()`*	Produces a Cohen-Friendly association plot indicating deviations from independence of the rows and columns in a two-way contingency table.
`cdplot()`*	Computes and plots the conditional densities describing how the conditional distribution of a categorical variable `y` changes over a numeric variable `x`.
`plot()`	Generic. The default method (`plot.default()`) produces a basic scatter plot between two continuous variables. Can also handle variables of POSIXct, POSIXlt, and Date class. Invoke `methods(plot)` to list all class specific methods.
`scatter.smooth()` (`stats`)	Generates an x-y scatterplot with smooth curves fitted by LOESS.
`persp()`*	Useful to graphically display three continuous variables. Generates an x-y scatterplot between two of the continuous variables and represents the third continuous variable by generating a 3D surface over the x-y plane.
`contour()`*	Useful to graphically display three continuous variables. Generates an x-y scatterplot between two of the continuous variables and represents the third continuous variable using contour lines. Alternative is the `filled.contour()` function.
`symbols()`*	Useful to graphically display three continuous variables. Generates an x-y scatterplot between two of the continuous variables and represents the third continuous variable using a symbol (e.g., circles of varying radius).
`image()`*	Useful to graphically display three continuous variables. Produces an x-y grid of rectangles and uses color to represent the value of a third variable z.
`pairs()`	Generates a matrix of scatterplots, plotting each continuous variables by all other continuous variables specified. Can be used in conjunction with the `panel.smooth()` function.
`stars()`*	Generates star (spider/radar) plots or segments diagrams of a multivariate data set.
`coplot()`*	Produces two types of conditioning plots.
`matplot()`	Plots the columns of one matrix against the columns of another.
`heatmap()`* (`stats`)	Draws a heat map, which is a false color image (basically `image(t(x))`), with a dendrogram added to the left side and top.
`sunflowerplot()`*	Useful when identical data values repeat a small number of times. Plots a 'flower' at each x-y location with a 'petal' for each replication of z.

'STANDARD' ARGUMENTS OF HIGH-LEVEL PLOTTING FUNCTIONS: Recall, in addition to function-specific arguments, there are several arguments that are 'standard' in the sense that many high-level plotting functions will accept them. Specifically, most high-level plotting functions will accept graphical parameter arguments that control such things as the appearance of axes and labels, and the range of the axes scales. It is usually possible to modify the default range of the axes scales on a plot by specifying the `xlim=` and/or `ylim=` arguments in the high-level function invocation, which are each specified as two element vectors containing the minimum and maximum values (e.g., `xlim = c(0, 50)`). There is also a set of arguments for modifying the default labels (if any) on a plot: `main=` for a title, `sub=` for a sub-title, `xlab=` for an x-axis label, and `ylab=` for a y-axis label. Each of these arguments is specified as a character string. The title specified with the `main=` argument (if any) is placed at the top of the plot in a large font, and the sub-title specified with the `sub=` argument is placed just below the x-axis in a smaller font. Some high-level plotting functions have an `axes=` argument (by default `TRUE`), which allows (if set to `FALSE`) you to suppress the drawing of the axes and therefore produce customized axes instead. Lastly, some high-level plotting function have an `add=` argument (by default `FALSE`), which, if set to `TRUE`, forces the function to act as a low-level plotting, superimposing the high-level plot onto the current plot.

Figure 4.1: Examples of high-level plotting functions – `mosaicplot()`, `spineplot()`, `fourfoldplot()`, and `assocplot()`.

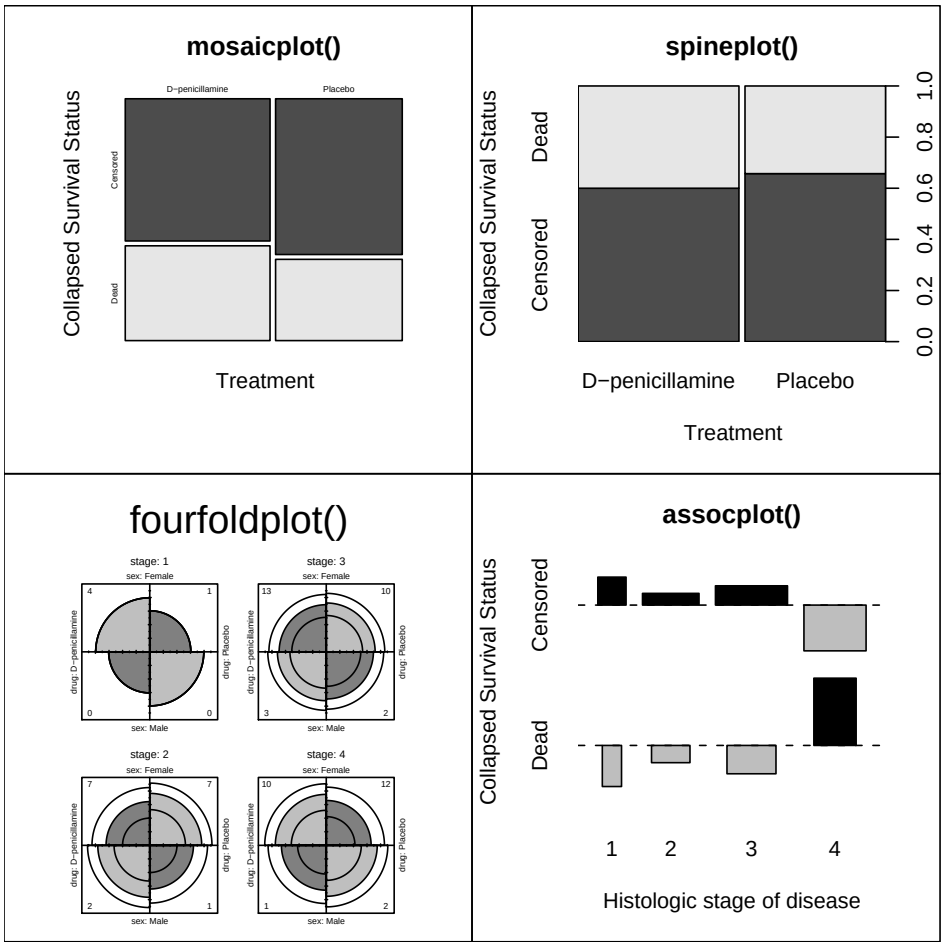


Figure 4.2: Examples of high-level plotting functions – `cdplot()`, `sunflowerplot()`, and `stars()`.

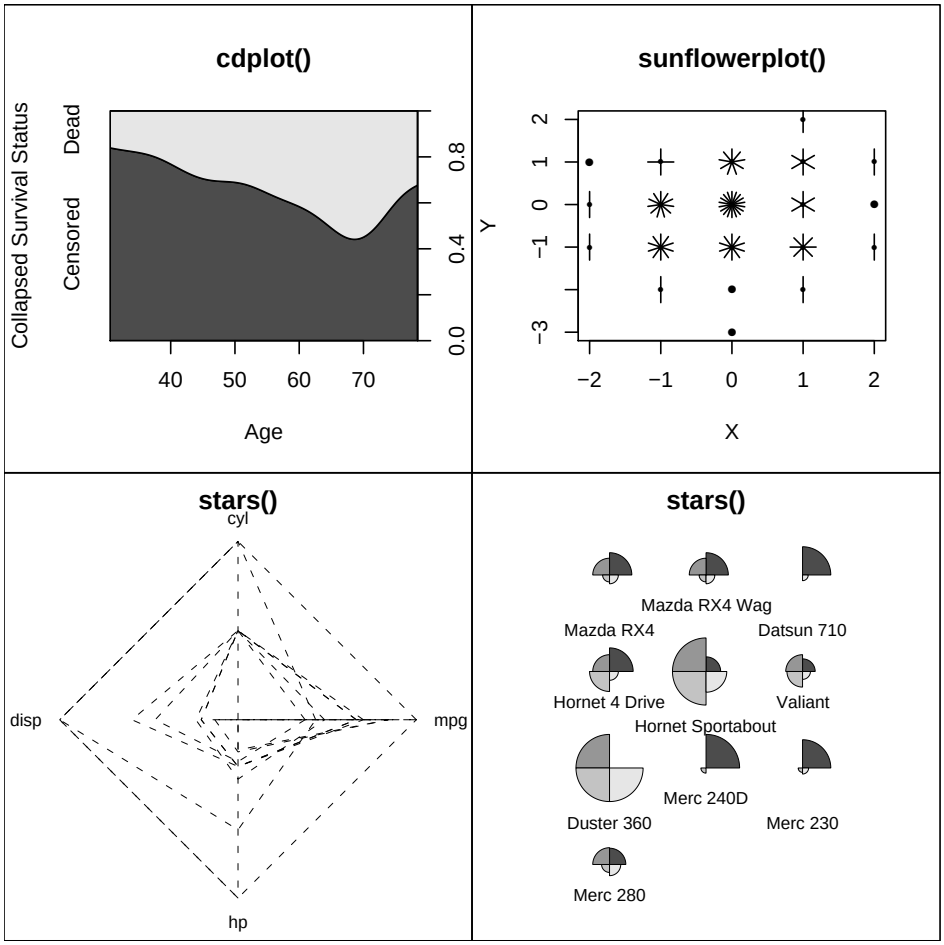


Figure 4.3: Examples of high-level plotting functions – `persp()`, `contour()`, `image()`, and `symbols()`.

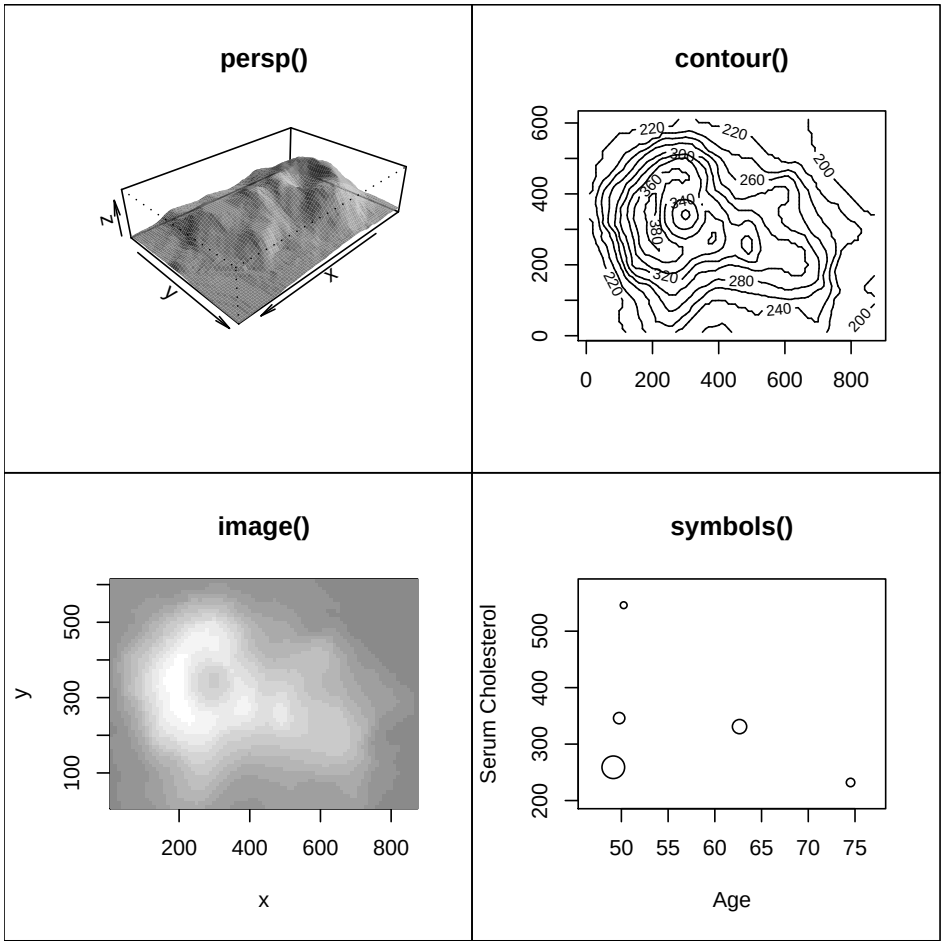


Figure 4.4: Examples of high-level plotting functions – `coplot()`.

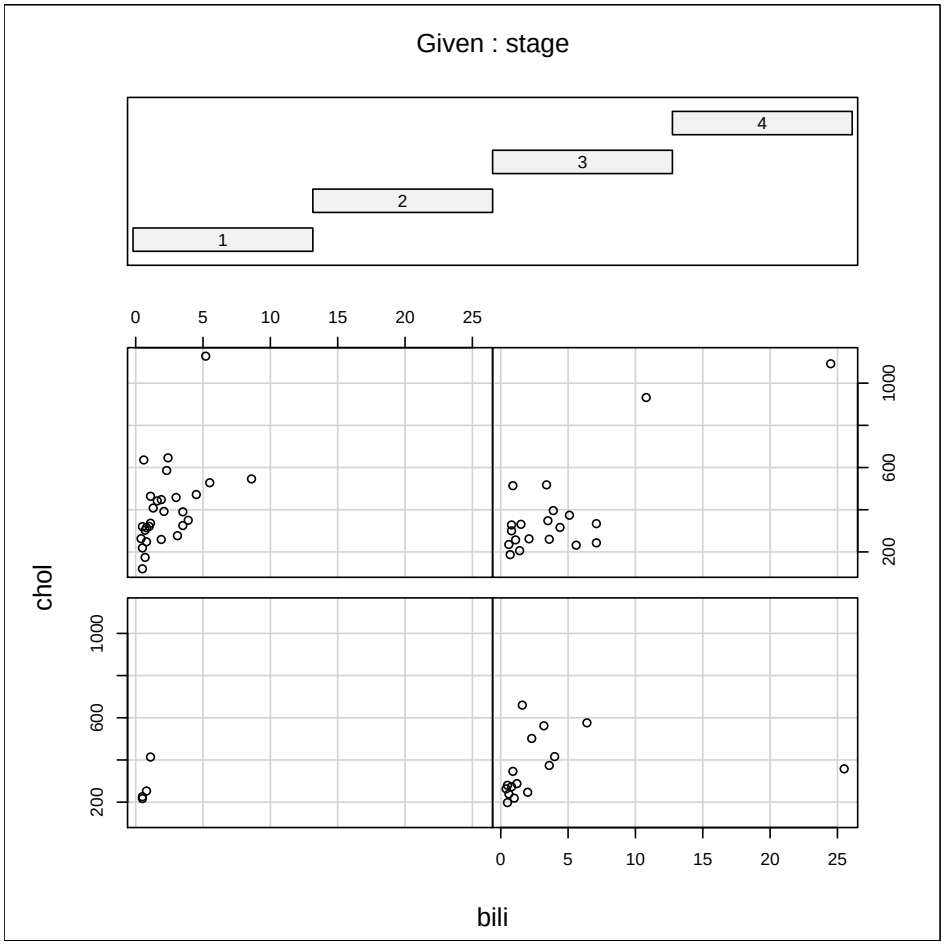
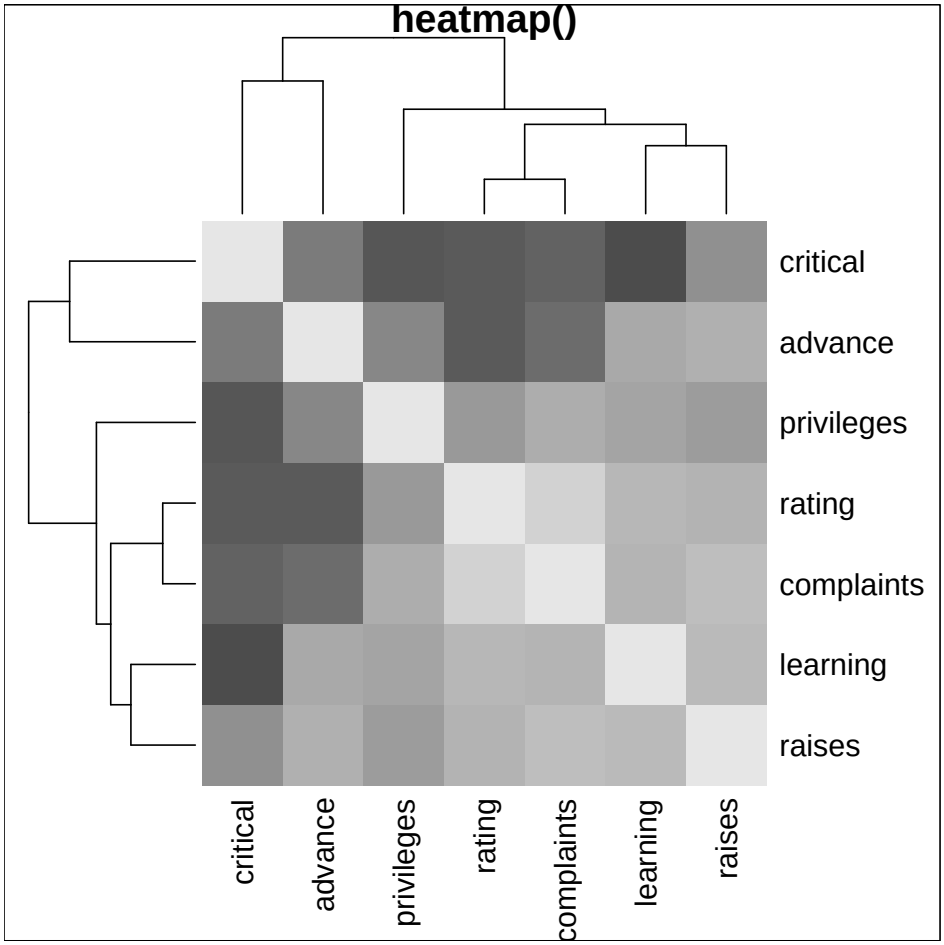


Figure 4.5: Examples of high-level plotting functions – `heatmap()`.



THE `par()` ***FUNCTION:*** Recall, the `par()` function is used to access and modify numerous graphical parameters. Also recall, that some of the `par()` function's arguments can be used as arguments to other high- and low-level plotting functions, while others can be queried and set *only* via the `par()` function. In addition, a small set of graphical parameters cannot be set at all and can only be queried using the `par()` function. The following table lists these subsets of the `par()` function's arguments. Note, each graphical parameter will be discussed in detail in the corresponding 'Section'.

Specification	Section	Parameter	Description
Queried and set via the `par()` function *and* used as arguments to other high- and low-level plotting functions	Points	`pch=`	data symbol type ('point character')
		`type=`	type of plot (points, lines, both)
	Lines	`lty=`	line type (solid, dashed)
		`lwd=`	line width
	Text	`adj=`	justification of text
		`ann=`	draw plot labels and titles?
		`cex=`	size of text (magnification multiplier)
		`cex.axis=`	size of axis tick labels
		`cex.lab=`	size of axis labels
		`cex.main=`	size of plot title
		`cex.sub=`	size of plot sub-title
		`font=`	font face (bold, italic) for text
		`font.axis=`	font face for axis tick labels
		`font.lab=`	font face for axis labels
		`font.main=`	font face for plot title
		`font.sub=`	font face for plot sub-title
		`las=`	rotation of text in margins
		`srt=`	rotation of text in plot region
		`tmag=`	size of plot title (relative to other labels)
	Color	`bg=`	'background' color
		`col=`	color of lines and data symbols
		`col.axis=`	color of axis tick labels
		`col.lab=`	color of axis labels
		`col.main=`	color of plot title
		`col.sub=`	color of plot sub-title
		`fg=`	'foreground' color
		`gamma=`	gamma correction for colors
	Axes	`bty=`	type of box drawn by the `box()` function
		`lab=`	number of ticks on axes
		`mgp=`	placement of axis title, tick labels, and line
		`tck=`	length of axis ticks (relative to plot size)
		`tcl=`	length of axis ticks (relative to text size)
		`xaxp=`	number of ticks on x-axis
		`xaxs=`	calculation of scale range on x-axis
		`xaxt=`	x-axis style (standard, none)
		`yaxp=`	number of ticks on y-axis
		`yaxs=`	calculation of scale range on y-axis
		`yaxt=`	y-axis style (standard, none)
	Plotting regions & Margins	`xpd=`	clipping region
Queried and set *only* via the `par()` function	Lines	`lend=`	line end style
		`ljoin=`	line join style
		`lmitre=`	line mitre limit
	Text	`family=`	font family for text
		`lheight=`	line spacing (multiplier)

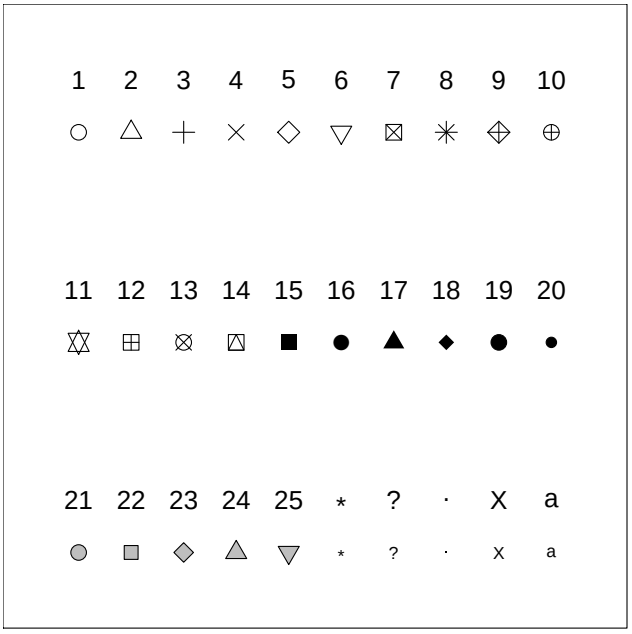
		ps=	size of text (points)
	Axes	usr=	range of scales on axes
		xlog=	logarithmic scale on x-axis?
		ylog=	logarithmic scale on y-axis?
	Plotting regions & Margins	fig=	location of figure region (normalized)
		fin=	size of region (inches)
		omd=	location of inner region (normalized)
		pin=	size of plot region (inches)
		plt=	location of plot region (normalized)
		pty=	aspect ratio of plot region
		mai=	size of figure margins (inches)
		mar=	size of figure margins (lines of text)
		mex=	line spacing in margins
		oma=	size of outer margins (lines of text)
		omi=	size of outer margins (inches)
	Multiple plots	ask=	prompt user before new page?
		mfcol=	number of figures on a page
		mfg=	which figure is used next
		mfrow=	number of figures on a page
	Overlaying output	new=	has a new plot been started?
Only queried using the `par()` function	Points	cin=	size of character (inches)
		cra=	size of character ('pixels')
		cxy=	size of character (user coordinates)
	Plotting regions & Margins	din=	size of graphics device (inches)

REMEMBER, invoking the `par()` function makes a persistent change to specific graphical parameter settings. Often, we want to modify some graphical parameters, do some plotting, and then restore the original graphics state. Using the `par()` function's `no.readonly=` argument, with no other arguments, returns only the parameters which can be set by a subsequent `par()` function call. This allows you to assign these returned parameters to an object, and, in turn, allows you to restore these initial parameters after making some modifications. For example,

```
op <- par(no.readonly = TRUE)
par(oma = c(7, 7, 10, 2), cex.main = 5, cex.lab = 2)
with(subset(pbc, drug == "Placebo"),
   plot(age.years ~ chol, main = "Age (years) vs. Serum Cholesterol",
   xlab = label(chol), ylab = label(age.years)))
par(op)
```

POINTS: R provides a fixed set of 26 data symbols for plotting points and the choice of data symbol is controlled by the `pch=` ('point character') graphical parameter, which is accepted as an argument to the `par()` function or as an argument to appropriate high- and low-level plotting functions. The `pch=` argument can be specified as either an integer value to select one of the fixed set of data symbols, or a single quoted character (e.g., either `pch = 19` or `pch = "+"`). If the `pch=` parameter is a character then that letter is used as the plotting symbol. The character '`.`' is treated as a special case and the device attempts to draw a very small dot. Figure 4.6 lists the available data symbols and their relevant integer value. In the future, the `Hmisc` package's `show.pch()` (invoked with no formal arguments), which plots the definitions of the `pch=` parameters, is also useful when you do not have Figure 4.6 handy. The color of the point character is controlled by the `col=` graphical parameter, and point characters 21 to 25 allow a fill color separate from the border color, with is controlled by the `bg=` (background) graphical parameter – see the 'Color' section.

Figure 4.6: The plotting symbols in R (`pch = 1:25`).

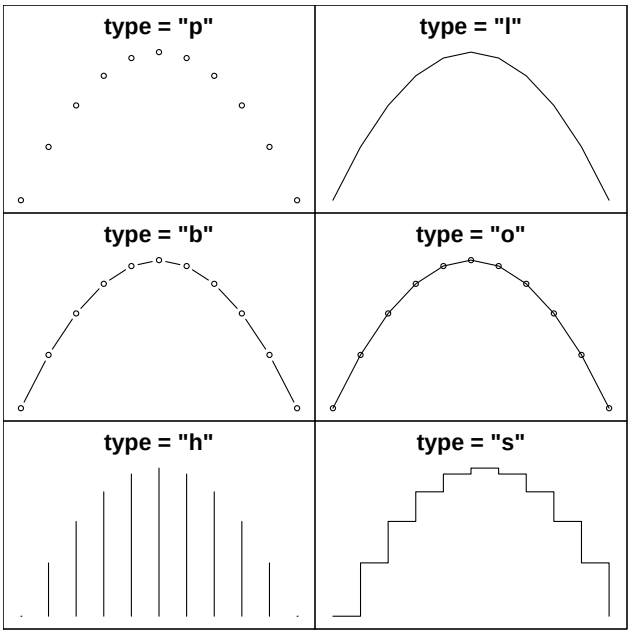


The size of the data symbol is linked to the size of the text and is affected by the `cex=` parameter. If the data symbol is a character, the size will also be affected by the `ps=` parameter – see the 'Text' section for more details.

The `type=` parameter controls how the data is represented in a plot (i.e., the type of plot produced). Unlike the `pch=`, the `type=` parameter is most often specified within a call to a high-level function (e.g., the `plot()` function) rather than via the `par()` function. Specifying `type = "p"` plots individual points at each `(x, y)` location (the default). Specifying `type = "l"` plots lines between each `(x, y)` location and individual points are *not* distinguished by a `pch=` data symbol. Specifying `type = "b"` plots both the data symbols distinguishing the points and the lines that connect them – the lines stop short of each data symbol. Like `type = "b"`, specifying `type = "o"` plots both the data symbols and the lines, but the lines *overlay* (i.e., 'over-plot') the points. Specifying `type = "h"` plots vertical lines from the x-axis to the `(x, y)` locations, which causes the plot to appear like a barplot or histogram with very thin bars. Two further values, `"s"` and `"S"` plot 'step-functions.' Specifically, specifying `type = "s"` plots lines going horizontally then vertically, implying the top of the vertical line defines the point. On the other hand, specifying `type = "S"` plots lines going vertically then horizontally, which implies the bottom of the vertical line defines the point. Finally, the value `"n"` (null) suppresses the plotting of any points and/or lines. However, axes are still drawn (by default) and the coordinate system is set up according to the data. Using `type = "n"` is ideal for crafting plots with subsequent low-level graphics functions. Figure 4.7 shows simple examples of the different plot types.

The main low-level plotting function used to add points to an existing plot is the `points()` function, which is a generic function that draws a sequence of points at the specified coordinates. If the data symbol is a character, the specified character(s) are centered at the coordinates when plotted. The `matpoints()` function can be used to add points to an existing `matplot()` function invocation. There are also several functions for helping to plot data when symbols overlap in a standard scatterplot, including the `jitter()` function. The `jitter()` function does no drawing but adds a very small amount of random noise to data values in order to separate values that are originally identical. The `Hmisc` package also has the `jitter2()` function, which does not add random noise, but retains unique values and ranks, and randomly spreads duplicate values at

Figure 4.7: Available basic plot types that can be specified with the `type=` graphical parameter. The relevant value of `type=` is shown above each plot.

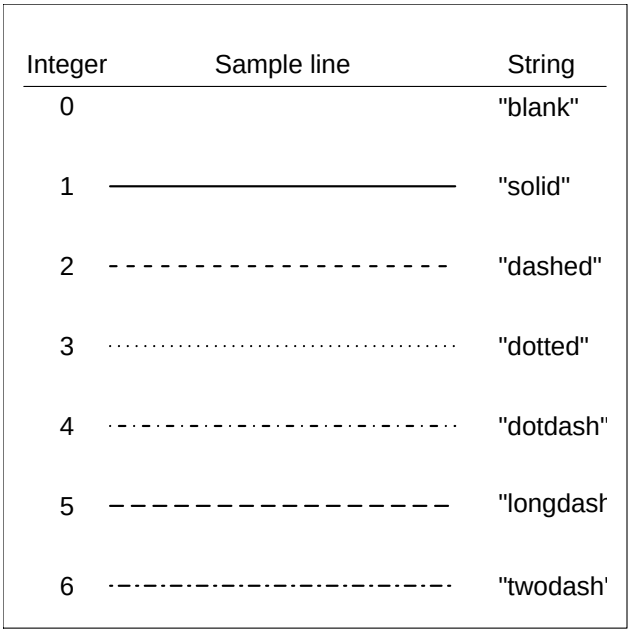


equidistant positions within limits of enclosing values.

LINES: There are five graphical parameters for controlling the appearance of lines. The `lty=` parameter describes the type of line to draw (solid, dashed, dotted, ...), which can be specified by name, such as "solid", or as an integer index. Figure 4.8 shows the available predefined line types. The `lwd=` parameter describes the width of the lines, which can be specified by a simple numeric value, e.g., `lwd = 3`. The interpretation of this value depends on what sort of device the line is being drawn on. In other words, the physical width of the line may be different when the line is drawn on a computer screen compared to when it is written to a file and then printed on a sheet of paper. And the `ljoin=`, `lend=`, and `lmitre=` parameters control how the ends and corners in the lines are drawn. When drawing thick lines, including rectangles and polygons, it becomes important to select the style that is used to draw corners (joins) in the line (`ljoin=`) and the ends of the line (`lend=`). R provides three styles for both cases. Specifically, the `ljoin=` parameter, which is used to control line joins, can be specified as `"mitre"` (pointy), `"round"`, or `"bevel"`. And the `lend=` parameter, which is used to control the line ends, can be specified as `"round"`, `"square"`, or `"butt"`. To avoid excessively pointy lines, the `ljoin=` parameter will be automatically converted from `ljoin = "mitre"` to `ljoin = "bevel"` if the angle at the join is too small. The point at which this automatic conversion occurs is controlled by the `lmitre=` parameter, which specifies the ratio of the length of the mitre divided by the line width. The default value is `10`, which means that the conversion occurs for joints where the angle is less than 11 degrees. Other standard values are `2`, which means that conversion occurs at angles less than 60 degrees, and `1.414`, which means that conversion occurs for angles less than 90 degrees. The minimum mitre limit value is `1`. The `lty=` and `lwd=` parameters are accepted as arguments to the `par()` function or as arguments to appropriate high- and low-level plotting functions. On the other hand, the `ljoin=`, `lend=`, and `lmitre=` parameters are accepted by the `par()` function only, and not all devices will respect them (especially `lmitre=`).

There are several low level functions that add lines to an existing plot. The `lines()` function is a generic function that joins the specified coordinates with line segments. The `matlines()` function can be used to add points to an existing `matplot()` function invocation. The `abline()` function adds one or more *straight* lines to the current plot, which can be specified using the intercept and slope (`a=` and `b=`, respectively, or

Figure 4.8: Available predefined line types that can be specified with the `lty=` graphics parameter.



as a 2 element vector with `coef=`) of the line, the y-value(s) for horizontal line(s) (`h=`), or the x-value(s) for vertical line(s) `v=`. The `arrows()` function draws 'arrows' between pair of points, which can be easily specified to draw error bars. Similarly, the `segments()` function draws line segments between pairs of points. The `grid()` function adds a `nx` by `ny` rectangular grid to an existing plot. The `rect()` function draws a rectangle (or sequence of rectangles) with the given coordinates, fill and border colors. The `polygon()` function draws the polygon whose vertices are given in the `x=` and `y=` arguments. And the `Hmisc` package offers three additional useful low-level plotting functions: (1) the `plsmo()` function adds a line to the existing plot that is a plot smoothed curve of `x` vs. `y`; (2) the `confbar()` function draws multi-level confidence bars using small rectangles that may be of different colors; and (3) the `errbar()` function adds vertical error bars to an existing plot or makes a new plot with error bars.

As an example of the usefulness of the `arrows()` function, let's look at age in years (`age.years`) across stage of disease (`stage`) where we plot the raw data as a stripchart and overlay an indication of the mean and standard deviations. The output is shown in Figure 4.9.

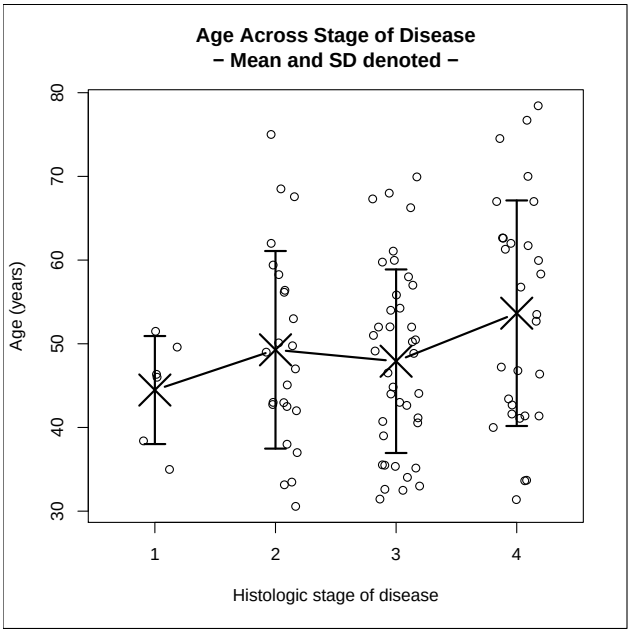
TEXT: There are a large number of traditional graphical parameters for controlling the appearance of text. There is also an `ann=` parameter, which indicates whether titles and axis labels should be drawn on a plot. This is intended to apply to high-level plotting functions, but is not guaranteed to work with all such functions.

JUSTIFICATION OF TEXT: Usually, the `adj=` parameter is a value of `0`, `0.5`, or `1` indicating the horizontal justification of text strings (`0` means left-justified, `1` means right-justified, and `0.5` centers text). However, the `adj=` parameter may also be specified as a two element numeric vector of the form `c(hjust, vjust)`, where `hjust` specifies the horizontal justification and `vjust` specifies vertical justification. The `adj=` parameter can be specified as an argument to the `par()` function or as an argument to appropriate high- and low-level plotting functions.

ROTATING TEXT: The `srt=` parameter specifies a rotation angle clockwise from the positive x-axis, in degrees. This will only affect text drawn in the plot region (i.e., text draw by the `text()` function). Unlike

Figure 4.9: Example of using the low-level `arrows()` function to add lines to an existing plot.

```
> xbar <- with(pbc, tapply(X = ageyrs, INDEX = stage, FUN = mean, na.rm = TRUE))
> sdev <- with(pbc, tapply(X = ageyrs, INDEX = stage, FUN = sd, na.rm = TRUE))
> with(pbc, stripchart(ageyrs ~ stage, method = "jitter", jitter = 0.2,
+     pch = 1, vertical = TRUE, col = "black", xlab = label(stage), ylab = paste(label(ageyrs),
+         " (", units(ageyrs), ")", sep = ""), main = paste("Age Across Stage of Disease",
+         "- Mean and SD denoted -", sep = "\n")))
> arrows(1:4, xbar + sdev, 1:4, xbar - sdev, angle = 90, code = 3, length = 0.1,
+     lwd = 2)
> lines(1:4, xbar, pch = 4, type = "b", cex = 4, lwd = 2)
> box("figure")
```



within the plot region, in the figure and outer margins, text may only be drawn at angles that are multiples of 90 degrees, and this angle is controlled by the `las=` parameter. A value of `0` means that text is always drawn parallel to the relevant axis (i.e., horizontal in margins 1 and 3, and vertical in margins 2 and 4). A value of `1` means text is always horizontal, `2` means text is always perpendicular to the relevant axis, and `3` means text is always vertical. This parameter interacts with or overrides the `adj=` and `srt=` parameters. Both the `srt=` and `las=` parameters can be specified as an argument to the `par()` function or as an argument to appropriate high- and low-level plotting functions.

TEXT SIZE: The size of text is ultimately a numerical value specifying the size of the font in 'points.' The font size is controlled by two parameters: the `ps=` parameter specifies an absolute font size parameter (e.g., `ps = 9`; can only be set via the `par()` function), and the `cex=` parameter specifies a multiplicative magnification modifier (e.g., `cex = 1.5`; a value < 1 shrinks the text and a value > 1 enlarges text). As with specifying color, the scope of a `cex=` parameter can vary depending on where it is given. When `cex=` is specified via the `par()` function, it affects most text. However, when `cex=` is specified via the `plot()` function, it only affects the size of the data symbols. Luckily, there are special parameters for controlling the size of text that is drawn as axis tick labels (`cex.axis=`), text that is drawn as axis labels (`cex.lab=`), text in the title (`cex.main=`), and text in the sub-title (`cex.sub=`). These four arguments can also be used as arguments in most high-level plotting functions. There is also a `tmag=` parameter for controlling the amount to magnify title text relative to other plot labels, which can also be used as an argument in most high-level plotting functions. Finally, the `strheight()` and `strwidth()` functions can be used to compute the height or width of character strings or mathematical expression, respectively, on the current plotting device in user coordinates, inches, or as fraction of the figure width.

MULTI-LINE TEXT: It is possible to draw text that spans several lines by inserting a new line escape sequence, `"\n"`, within a piece of text. For example, `"first line\nsecond line"`. The spacing between lines is controlled by the `lheight=` parameter, which is a multiplier added to the natural height of a line of text (e.g., `lheight = 2` specifies double-space text), and can only be specified via the `par()` function.

SPECIFYING FONTS: The font face is specified via the `font=` parameter as an integer, and can be specified as an argument to the `par()` function or as an argument to appropriate high- and low-level plotting functions. `font = 1` produces Roman or upright face; `font = 2` produces bold face font; `font = 3` produces slanted or italic face; and `font = 4` produces bold and slanted face. As with color and text size, the `font=` parameter applies only to text drawn in the plot region. There are also additional parameters specifically for axes (`font.axis=`), labels (`font.lab=`), and titles (`font.main=` and `font.sub=`). These four arguments can also be used as arguments in most high-level plotting functions. Also, every graphics device establishes a default font family, which is usually a sans serif font such as Helvetica or Arial. A new font family is specified using the `family=` parameter via the `par()` function.

In addition to all of these graphical parameters, there are a large number of low-level plotting functions that control the placement of text onto your plot. For instance, the `text()` function draws the supplied text strings at specified coordinates within the plot region. The `mtext()` function writes text in one of the four margins of the current figure region or one of the four outer margins of the device region – see the 'Plotting regions & Margins' section for more details. And the `title()` function can be used to add labels to a plot – the main plot title (the `main=` argument; placed at top), the plot sub-title (the `sub=` argument; placed at bottom), the x-axis label (the `xlab=` argument), and the y-axis label (the `ylab=` argument). These four arguments can also be used as arguments in most high-level plotting functions.

There is also the `identify()` function, which can be used to *interactively* add labels to data symbols on a plot. The `identify()` function performs no plotting itself, but simply allows the user to move the mouse pointer and click the left mouse buttton near a point. If there is a point near the mouse pointer it will be marked with its index number (that is, its position in the `x`/`y` vectors) plotted nearby. Alternatively, you could use some informative string (such as a case name) as a highlight by using the `identify()` function's `labels=` argument, or disable marking altogether with the `plot = FALSE` argument. When the process is terminated, the `identify()` function returns the indices of the selected points; you can use these indices to

extract the selected points from the original vectors `x` and `y`.

COLOR: There are three main color parameters: `col=`, `fg=`, and `bg=`. The `col=` parameter is the most commonly used. The primary use is to specify the color of points, lines, text, and so on that are drawn in the plot region. Unfortunately, when specified via a high- and/or low-level plotting function, the effect can vary. For example, a standard scatterplot produced by the high-level `plot()` function will use the `col=` parameter for coloring data symbols and lines, but the high-level `barplot()` function will use the `col=` parameter for filling the contents of its bars. In the low-level `rect()` function, the `col=` parameter provides the color to fill the rectangle and there is a `border=` argument specific to the low-level `rect()` function that gives the color to draw the border of the rectangle. The effect of the `col=` parameter on graphical output drawn in the margins also varies. It does not affect the color of axes and axis labels, but it does affect the output from the low-level `mtext()` function. There are also specific parameters for affecting axes, labels, titles, and subtitles called `col.axis=`, `col.lab=`, `col.main=`, and `col.sub=`, respectively. The `fg=` parameter is primarily intended for specifying the color of axes and borders on plots. There is some overlap between this and the specific `col.axis=`, `col.main=`, etc parameters mentioned above. The `bg=` parameter is primarily intended to specify the color of the background for graphical output. This color is used to fill the entire page. As with the `col=` parameter, when `bg=` is specified in a graphics function it can have a quite different meaning. For example, the high-level `plot()` function and the low-level `points()` function use the `bg=` parameter to specify the color for the interior of the points, which can have different colors on the border (i.e., `pch=` parameter values `21` to `25`; see the 'Points' section). There is also the `gamma=` parameter that controls the gamma correction for a device. On most devices this can only be set once the device is first opened.

SPECIFYING COLOR: The easiest way to specify a color in R is simply to use the color's name. For example, `"red"` can be used to specify that graphical output should be (a very bright) red. R understands a fairly large set of color names (657 to be exact), which can be returned by the `colors()` (or `colours()`) functions (invoked with no formal arguments). The following function generates a matrix of rectangles that displays the corresponding color of each of the 657 possible colors you can specify with the `colors()` function.

```
show.colors <- function(x = 22,y = 30){
   par(mar = c(0.1, 0.1, 0.1, 0.1))
   plot(c( -1, x), c(-1, y), xlab = "", ylab = "", type = "n", xaxt = "n", yaxt = "n", bty = "n")
   for(i in 1:x) {
      for(j in 1:y){
         k <- y*(i - 1) + j ; co <- colors()[k]
         rect(i - 1, j - 1, i, j, col = co, border = 1)}}
   text(rep(-0.5, y), (1:y) - 0.5, 1:y, cex = 1.2 - 0.016*y)
   text((1:x) - 0.5, rep(-0.5, x), y*(0:(x-1)), cex = 1.2 - 0.022*x)
}
show.colors()
```

By default, R has a color palette of eight colors, which can be returned by the `palette()` function (invoked with no formal arguments):

```
> palette()

[1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"  "gray"
```

In addition to being able to specify these eight colors by name, these eight colors can also be specified by integer values (i.e. `1` through `8`). The `Hmisc` package's `show.col()` function plots the definitions of these eight integer-value colors in case you forget. It is also possible to specify the colors using one of the standard color-space descriptions. For example, the `rgb()` function allows a color to be specified as a Red-Green-Blue (RGB) triplet of intensities. For example, using this function, the color red is specified as `rgb(1, 0, 0)` (i.e., as much red as possible, no blue, and no green). The `col2rgb()` function can be used to see the RGB values for a particular color name. An alternative way to provide an RGB color specification is to provide a string of the form `"#RRGGBB"`, where each of the pair `RR`, `GG`, and `BB` consist of two hexadecimal digits giving a value in the range zero (00) to 255 (FF). In this specification, the color red is given as `"#FF0000"`. There

is also an `hsv()` function for specifying a color as a Hue-Saturation-Value (HSV) triplet. The terminology of color spaces is fraught, but roughly speaking: hue corresponds to a position of the rainbow, from red (0), through orange, yellow, green, blue, indigo, to violet (1); saturation determines whether the color is dull or bright; and value determines whether the color is light or dark. The HSV specification for the (very bright) color red is `hsv(0, 1, 1)`. The `rgb2hsv()` function converts a color specification from RGB to HSV.

COLOR SETS: More than one color is often required within a single plot and in such cases it can be difficult to select colors that are aesthetically pleasing or are related in some way (e.g., a set of color in which the brightness of the colors decreases in regular steps). The following table lists some functions that R provides for generating sets of colors. Each of the functions listed selects a set of colors by taking regular steps along a path through the HSV color space. Also, the `colorRamp()` and `colorRampPalette()` functions can be used to interpolate a new color set from an existing set of colors.

Name	Description
`rainbow()`	Colors vary from red through orange, yellow, green, blue, and indigo, to violet.
`heat.colors()`	Colors vary from white, through orange, to red.
`terrain.colors()`	Colors vary from white, through brown, to green.
`topo.colors()`	Colors vary from white, through brown then green, to blue.
`cm.colors()`	Colors vary from light blue, through white, to light magenta.
`gray()`	A set of shades of gray.
`gray.colors()`	A set of gamma-corrected gray colors.

AXES: By default, the traditional graphics system produces axes with sensible labels and tick marks at sensible locations. However, if the axis does not look right, there are a number of graphical parameters specifically for controlling aspects such as the number of tick marks and the positioning of labels. Recall, most high-level plotting functions provide `xlim=` and `ylim=` arguments to control the range of the scale on the axes. If none of these gives the desired result, you may have to resort to drawing the axis explicitly using the low-level `axis()` plotting function – more to come.

The `lab=` parameter in the traditional graphics state is used to control the number of tick marks on the axes. The parameter is only used as a starting point for the algorithm R uses to determine sensible tick mark locations so the final number of tick marks that are drawn could easily differ from this specification. The parameter takes two values: the first specifies the number of tick marks on the x-axis and the second specifies the number of tick marks on the y-axis.

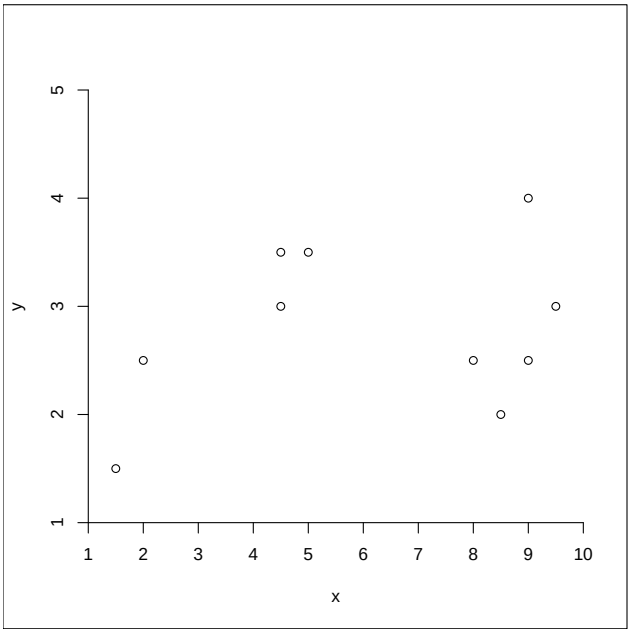
The `xaxp=` and `yaxp=` parameter also relate to the number and location of the tick marks on the axes of a plot. This parameter is almost always calculated by R for each new plot so user parameters are usually overridden. In other words, it only makes sense to query this parameter for its current value. The parameters consist of three values: the first two specify the location of the left-most and right-most tick-marks (bottom and top tick-marks for the y-axis), and the third value specifies how many intervals there are between tick marks. However, when a log transformation is in effect for an axis, the three values have a different meaning altogether (see the `par()` function's help file).

The `mgp=` parameter controls the distance that the components of the axes are drawn away from the edge of the plot region. There are three values representing (in order) the positioning of the axis label, the tick mark labels, and the axis line. The values are in terms of lines of text away from the edges of the plot region. By default `mgp = c(3, 1, 0)`.

The `tck=` and `tcl=` parameters control the length of the tick marks. The `tcl=` parameter specifies the length of tick marks as a fraction of the height of a line of text. The sign dictates the direction of the tick marks – a negative value draws tick marks outside the plot region and a positive value draws tick marks inside the plot

Figure 4.10: Specifying the extremes of the coordinates of the plotting region with the `par()` function's `usr=` argument.

```
> plot(0, 0, type = "n", axes = FALSE, xlab = "x", ylab = "y")
> par(usr = c(1, 10, 1, 5))
> axis(side = 1, at = 1:10)
> axis(side = 2, at = 1:5)
> points(x = sample(seq(1.5, 9.5, by = 0.5), size = 10, replace = TRUE),
+     y = sample(seq(1.5, 4.5, by = 0.5), size = 10, replace = TRUE))
> box("figure")
```



region. The `tck=` parameter specifies tick mark lengths as a fraction of the smaller of the physical width or height of the plotting region, but it is only used if its value is not `NA` (by default, `tck = NA`). An alternative to modifying the `tck=` and/or `tcl=` parameters is `Hmisc` package's low-level `minor.tick()` function, which adds minor (shorter) tick marks to an existing plot.

The `xaxs=` and `yaxs=` parameters control the 'style' of the axes of a plot. By default, the parameter is `"r"`, which means that R calculates the range of values on the axis to be wider than the range of the data being plotted (so that data symbols do not collide with the boundaries of the plot region). It is possible to make the range of the values on the axis exactly match the range of values in the data by specifying the value `"i"`. This can be useful if the range of values on the axes are being explicitly controlled via the `xlim=` or `ylim=` arguments to a high-level plotting function.

An alternative to the `xaxs=` and `yaxs=` parameters is the `usr=` parameter, which will allow us to specify that the two axes should intersect at an exact x-y coordinate. Specifically, the `usr=` parameter is specified as a vector of the form `c(x1, x2, y1, y2)` giving the extremes of the coordinates of the plotting region. For example, let's generate a plot such that the x- and y- axes intersect at (1, 1) – see Figure 4.10. Notice, with the `usr=` parameter, you have to start a new plot first and *then* you set the `usr=` parameter. Also, notice that we specified `type = "n"` and `axes = FALSE` in the `plot()` function invocation and then added the axes back in with the `axis()` function.

The `xaxt=` and `yaxt=` parameters control the 'type' of axes. The default value, `"s"`, means that the axis is drawn. Specifying a value of `"n"` means that the axis is not drawn.

The `xlog=` and `ylog=` parameters control the transformations of the values on the axes. The default value is `FALSE`, which means that the axes are linear and values are not transformed. If this value is `TRUE` then a logarithmic transformation is applied to any values on the relevant dimension in the plot region. This also affects the calculation of tick mark locations on the axes.

The `bty=` parameter is not strictly to do with axes, but it controls the type of 'box' that is drawn around a plot. The value can be `"n"`, which means that no box is drawn, or it can be one of `"o"`, `"l"`, `"7"`, `"c"`, `"u"`, or `"]"`, which means that the box drawn resembles the corresponding uppercase character. For example, `bty = "c"` means that the bottom, left, and top borders will be draw, but the right border will not be drawn. An alternative to using the `bty=` parameter is to use the low-level `box()` function. With the `box()` function you can specify the box-type (`bty=`), color (`col=`), line type (`lty=`), and line width (`lwd=`) of the box drawn. You can also specify *which* (`which=`) box to draw. By default, the `box()` function will draw a box connecting all four axes (`which = "plot"`). However, `which=` can be changed to `"figure"`, `"inner"`, or `"outer"`, which come in very handy when you want to outline the wholeplot or specific plots when plotting multiple plots.

In some case, it may be useful to draw tick marks at the locations that the default axis would use, but with different labels. The `axTicks()` function can be used to calculate these default locations. This function is also useful for enforcing an `xaxp=` (or `yaxp=`) graphical parameter. If these parameters are specified via the `par()` function, they usually have no effect because the traditional graphics system almost always calculates the parameter itself. You can choose these parameters by passing them as arguments to the `axTicks()` function and then passing the resulting locations via the `at=` argument to the `axis()` function. The `pretty()` function is another useful function for determining where to place the tick marks.

THE `axis()` FUNCTION: When using the `axis()` function to draw an axis 'from scratch,' the first step is to inhibit the default axes. Most high-level plotting functions should provide an `axes=` argument which, when set to `FALSE`, indicates that the axes should not be drawn. Specifying the graphical parameter `xaxt = "n"` (or `yaxt = "n"`) or `ann = FALSE` via the `par()` function may also do the trick. Once this is done, the `axis()` function can draw axes on any side of the plot (specified by the `side=` argument), and you can specify the location along the axis of tick marks and the text to use for tick labels using the `at=` and `labels=` arguments, respectively. See Figure 4.11 for an example. The `axis()` function is not generic, but there are special alternative functions for plotting time related data. Specifically, the `axis.Date()` and `axis.POSIXct()` functions take an object containing dates and produce an axis with appropriate labels representing times, days, months, and years (e.g., `10:15`, `Jan 12`, or `1995`). In addition, the `axis()` function is useful to superimpose the underlying numerical axes that are used by functions like `dotchart()`, `stripchart()`, `boxplot()`, or `barplot()` – use `axis(side = 1)` to superimpose the x-axis and `axis(side = 2)` for the y-axis. This allows you to determine the x-y location of specific things in the plot, such as where to place a legend or the middle of a bar in order to add text. Lastly, the `axis()` function can be used to draw text in the margins – see Figure 4.12 for an example.[1] It is important to realize that the `axis()` function will not allow you to place any text in the margin areas where two axes intersect. An alternative is to use the `mtext()` function (discussed in the 'Text' section) with its `line=` and `at=` arguments.

PLOTTING REGIONS & MARGINS: In the base graphics system, every page is split up into three main regions: (1) the *outer margins*; (2) the current *figure region*; and (3) the current *plot region*. Figure 4.13 shows these regions when there is only one figure on the page and Figure 4.14 shows the regions when there are multiple figures on the page. The region obtained by removing the outer margins from the device is called the *inner* region. When there is only one figure, this usually corresponds to the figure region, but when there are multiple figures the inner region corresponds to the union of all figure regions. The area outside the plot region, but inside the figure region is referred to as the *figure margins*. A typical high-level function draws data symbols and lines within the plot region and axes and labels in the figure or outer

[1] The `rates.dat` file contains a data set for a study of beta-blocker adherence post-AMI.

Figure 4.11: Example of customizing axes using the low-level `axis()` plotting function.

```
> op <- par(read.only = TRUE)
> x <- 1:2
> y <- runif(n = 2, min = 0, max = 100)
> par(cex = 0.8, mar = c(4, 5, 2, 5))
> plot(x, y, type = "n", xlim = c(0.5, 2.5), ylim = c(-10, 110), axes = FALSE,
+     ann = FALSE)
> axis(side = 2, at = seq(from = 0, to = 100, by = 20))
> mtext("Temperature (Centigrade)", side = 2, line = 3)
> axis(side = 1, at = 1:2, labels = c("Treatment 1", "Treatment 2"))
> axis(side = 4, at = seq(from = 0, to = 100, by = 20), labels = seq(from = 0,
+     to = 100, by = 20) * 9/5 + 32)
> mtext("Temperature (Fahrenheit)", side = 4, line = 3)
> box()
> segments(x, 0, x, 100, lwd = 20, col = "dark grey")
> segments(x, 0, x, 100, lwd = 16, col = "white")
> segments(x, 0, x, y, lwd = 16, col = "light grey")
> box("figure")
> par(op)
```

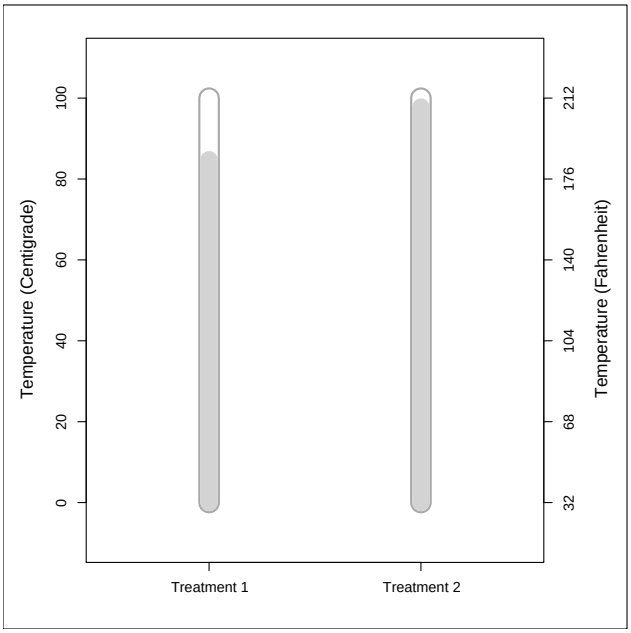


Figure 4.12: Example of customizing axes and margins using the low-level `axis()` plotting function and other low-level text plotting functions.

```
> rates <- read.table("rates.dat", header = TRUE)
> op <- par(no.readonly = TRUE)
> par(mar = c(10, 5, 4, 2), bg = "white")
> plot(rates$day, rates$rate1, type = "l", ylim = c(0, 85), axes = FALSE,
+     xlab = "Days since discharge", ylab = "Percent beta-blocker users (%)")
> axis(1, at = c(0, 30, 90, 180, 270, 365))
> axis(2, at = c(0, 20, 40, 60, 80))
> axis(1, at = rates$day, labels = rates$atrisk1, tick = FALSE, line = 4,
+     cex = 0.8)
> axis(1, at = rates$day, labels = rates$atrisk0, tick = FALSE, line = 6.5,
+     cex = 0.8)
> lines(rates$day, rates$rate0, type = "l")
> box()
> mtext("No. at-risk: patients discharged on beta-blockers", side = 1,
+     line = 4, adj = 0, cex = 0.8)
> mtext("No. at-risk:  patients not discharged on beta-blockers", side = 1,
+     line = 6.5, adj = 0, cex = 0.8)
> mtext("Outpatient adherence to\nbeta-blocker therapy post-AMI", side = 3,
+     cex = 1.2, line = 1)
> text(180, 70, "Discharged on beta-blockers", cex = 0.8)
> text(180, 20, "Not discharged on beta-blockers", cex = 0.8)
> box("outer")
> par(op)
```

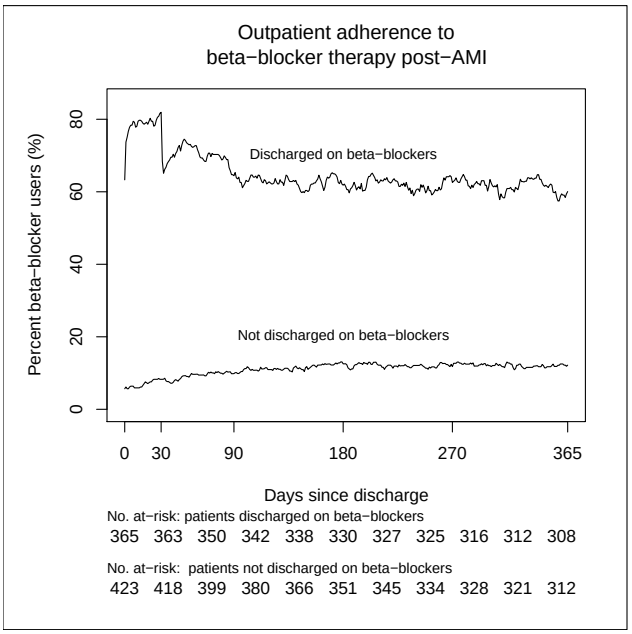
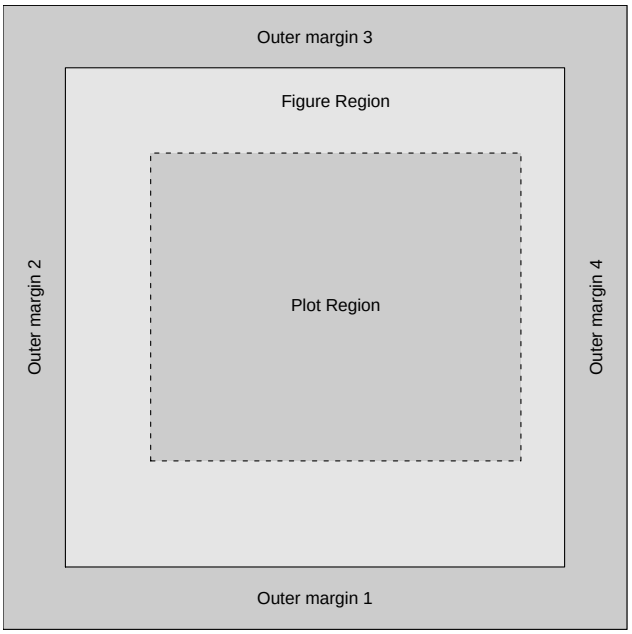


Figure 4.13: The outer margins, figure region, and plot region when there is a single plot on the page.



margins. The size and location of the different regions are controlled either via the `par()` function, or using special functions for arranging multiple plots – see the 'Multiple plots' section. Specifying an arrangement of the regions does not usually affect the current plot as the parameters only come into effect when the next plot is started.

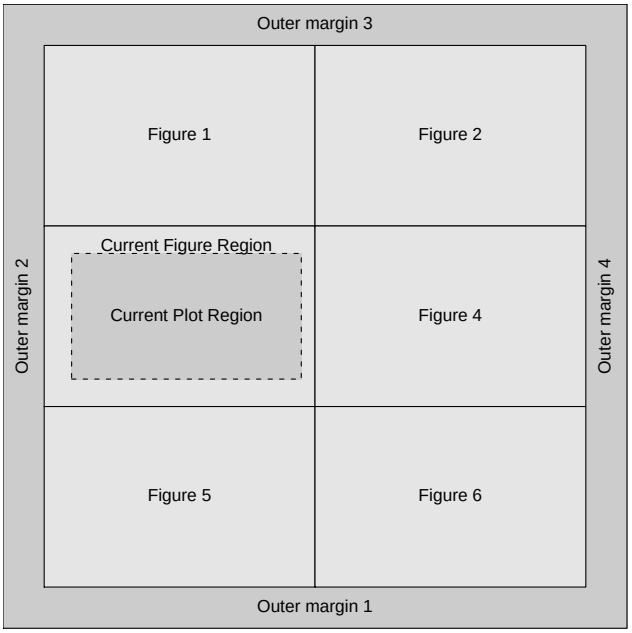
OUTER MARGINS: By default, there are not outer margins on a page. Outer margins can be specified using the `oma=` graphical parameter. This consists of four values for the four margins in the order `(bottom, left, top, right)` and values are interpreted as lines of text (a value of 1 provides space for one line of text in the margin). The margins can also be specified in inches using the `omi=` parameter or in normalized device coordinates (i.e., a proportion of the device region) using the `omd=` parameter. In the latter case, the margins are specified in the order `c(left, right, bottom, top)`. Recall, the `oma=`, `omi=`, and `omd=` parameters can only be set using the `par()` function.

FIGURE REGIONS: By default, the figure region is calculated from the parameters for the outer margins, and the number of figures in the page. The figure region can be specified explicitly using either the `fig=` or `fin=` parameters via the `par()` function. The `fig=` parameter specifies the location, `c(left, right, bottom, top)`, of the figure region where each value is a proportion of the 'inner' region (the page less the outer margins). The `fin=` parameter specifies the size, `(width, height)`, of the figure region in inches and the resulting figure region is centered within the inner region.

FIGURE MARGINS: The figure margins can be controlled using the `mar=` parameter. This consists of four values for the four margins in the order `c(bottom, left, top, right)` where each value represents a number of lines of text. The default is `c(5, 4, 4, 2) + 0.1`. The margins may also be specified in terms on inches using the `mai=` parameter. Remember, the `mar=` and `mai=` parameters can only be set using the `par()` function.

The `mex=` parameter controls the size of a 'line' in the margins, and can only be set using the `par()` function. The `mex=` parameter does not affect the size of text drawn in the margins, but is used to multiply the size of text to determine the height of one line of text in the margins.

Figure 4.14: The outer margins, *current* figure region, and *current* plot region when there are multiple plots on the page.



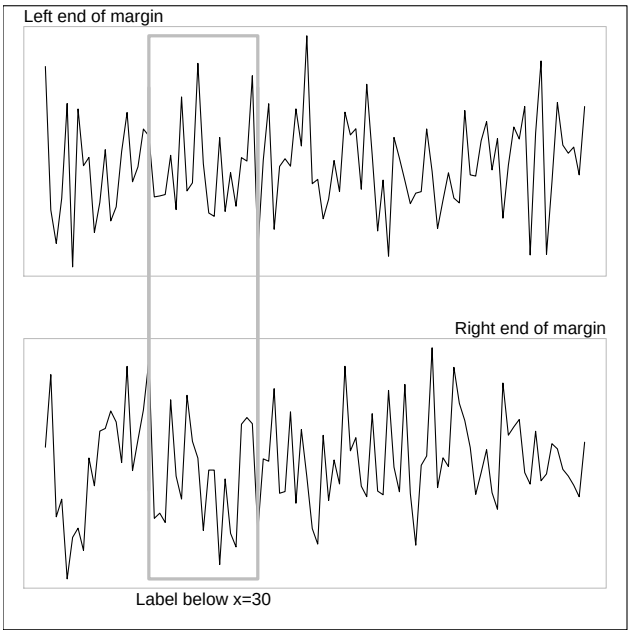
By default, the plot region is calculated from the figure region less the figure margins. The location and size of the plot region may be controlled explicitly using the `plt=`, `pin=`, or `pty=` parameters via the `par()` function. The `plt=` parameter allows you to specify the location of the plot region, `c(left, right, bottom, top)`, here each value is a proportion of the current region. The `pin=` parameter specifies the size of the plot region, `c(width, height)`, in terms of inches. The `pty=` parameter controls how much of the available space (figure region less figure margins) that the plot region occupies. The default value is `"m"`, which means that the plot region occupies all of the available space. A value of `"s"` means that the plot region will take up as much of the available space as possible, but it must be 'square' (i.e., its physical width will be the same as its physical height).

CLIPPING: Traditional graphical output is usually clipped to the plot region. This means that any output that would appear outside the plot region is not drawn. In addition, traditional high- and low-level plotting functions that draw in the margins clip output to the current figure region or to the device. Obviously, it can be useful to override the default clipping region. For example, this is necessary to draw a legend outside the plot region using the `legend()` function – see the 'Other additions' section. The traditional clipping region is controlled via the `xpd=` parameter, and is specified via the `par()` function. Clipping can occur either to the whole device (`xpd = NA`), to the current figure region (`xpd = TRUE`), or to the current plot region (`xpd = FALSE, the default`). As an example, let's demonstrate how to annotate the margins of a traditional graphics plot – see Figure 4.15.

MULTIPLE PLOTS: There are a number of ways to produce multiple plots on a single page. Specifically, the number of plots on a page and their placement on the page can be controlled directly by specifying the `mfrow=` (or `mfcol=`) parameters using the `par()` function, or through a higher-level interface provided by the `layout()` function. In addition, the `split.screen()` function (and associated functions) provide another approach where a figure region can itself be created as a complete page to split into further figure and plot regions. We will discuss the `split.screen()` function in 'Graphical output' section.

Figure 4.15: Annotating the margins of a traditional graphics using the `xpd=` (clipping) parameter via the `par()` function. Text has been added in the top margin of the upper plot and in the top and bottom margin of the lower plot, and thick gray lines have been added to both plots (and overlapped so that it appears to be a single rectangle across both plots).

```
> y1 <- rnorm(100)
> y2 <- rnorm(100)
> par(mfrow = c(2, 1), mar = c(2, 1, 1, 1), xpd = NA)
> plot(y1, type = "l", axes = FALSE, xlab = "", ylab = "", main = "")
> box(col = "grey")
> mtext("Left end of margin", adj = 0, side = 3)
> lines(x = c(20, 20, 40, 40), y = c(-7, max(y1), max(y1), -7), lwd = 3,
+     col = "grey")
> plot(y2, type = "l", axes = FALSE, xlab = "", ylab = "", main = "")
> box(col = "grey")
> mtext("Right end of margin", adj = 1, side = 3)
> mtext("Label below x=30", at = 30, side = 1)
> lines(x = c(20, 20, 40, 40), y = c(7, min(y2), min(y2), 7), lwd = 3,
+     col = "grey")
> box("outer")
```



These three approaches are mutually incompatible. For example, a call to the `layout()` function will override any previous `mfrow=` or `mfcol=` parameters. Also, some high-level functions (e.g., the `coplot()` function) call the `layout()` and `par()` functions themselves to create a plot arrangement, which means that the output from such functions cannot be arranged with other plots on a page.

THE `par()` FUNCTION APPROACH: The number of figure regions on a page can be controlled via the `mfrow=` and `mfcol=` graphical parameters. Both of these consist of two values indicating a number of rows, *nr*, and a number of columns, *nc* (i.e., `mfrow = c(nr, nc)`); these parameters result in the *nr* x *nc* figure regions of *equal* size. For example, the code `par(mfrow = c(3,2))` creates six figure regions on the page arranged in three rows and two columns. The top left region figure is used first. If the parameter is made via the `mfrow=` parameter then the figure regions along the top row are used next from left to right, until that row is full. After that, figure regions are used in the next row down, from left to right, and so on. When all rows are full, a new page is started. If the parameter is made via the `mfcol=` parameter, figure regions are used by column instead of by row. We demonstrated the use of the `mfrow=` parameter in the first document. The order in which figure regions are used can be controlled using the `mfg=` parameter to specify the next figure region. This parameter consists of two values that indicate the row and column of the next figure to use. In addition, the `ask=` parameter controls whether the user is prompted before the graphics system starts a new page of output – use `ask = TRUE` to be prompted. It is useful for viewing multiple pages of output (e.g., the output from `example(boxplot)`) that otherwise flick by too fast to view properly.

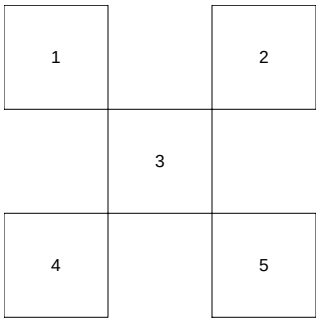
THE `layout()` FUNCTION APPROACH: The `layout()` function provides an alternative to the `mfrow=` and `mfcol=` parameters. The primary difference is that the `layout()` function allows the creation of multiple figure regions of *unequal* sizes. The simple idea underlying the `layout()` function is that it divides the inner region of the page into a number of rows and columns, but the heights of the rows and the widths of the columns can be independently controlled, *and* a figure can occupy more than one row or more than one column. In addition, one or more intersections of the rows and columns can be left blank.

The first argument (and the only required argument) to the `layout()` function is a matrix. The number of rows and columns in the matrix determines the number of rows and columns in the layout. The contents of the matrix are integer values that determine which rows and columns each figure will occupy. A value of `0` can also be used, which means that no figure will occupy the specified region. For example, the following layout specification is identical to `par(mfrow = c(3, 2))`: `layout(matrix(c(1, 2, 3, 4, 5, 6), byrow = TRUE, ncol = 2))`. We can visualize the the partitions created using the `layout.show()` function – we merely specify the number of figures to plot. For example,

```
> (m <- matrix(c(1, 0, 2, 0, 3, 0, 4, 0, 5), byrow = TRUE, ncol = 3))

     [,1] [,2] [,3]
[1,]    1    0    2
[2,]    0    3    0
[3,]    4    0    5

> layout(m)
> layout.show(9)
```



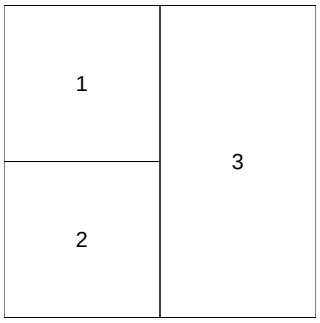
The contents of the layout matrix determine the order in which the resulting figure regions will be used. For example, the following code creates a layout with exactly the same rows and columns as the previous one, but the figure regions will be used in the reverse order: `layout(matrix(c(6, 5, 4, 3, 2, 1), byrow = TRUE, ncol = 2))`.

A figure may also occupy more than one row or column in the layout. For example, in the layout specified by the code `layout(matrix(c(1, 2, 3, 3), byrow = FALSE, ncol = 2))`, the third plot will fill a figure region that occupies both rows of the second column.

```
> (m <- matrix(c(1, 2, 3, 3), byrow = FALSE, ncol = 2))

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

> layout(m)
> layout.show(4)
```

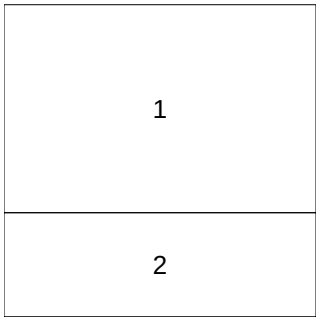


By default, all row heights are the same and all column widths are the same size and the available inner region is divided up equally. The `heights=` argument can be used to specify that certain rows are given greater portion of the available height (for all of what follows, the `widths=` arguments works analogously for the column width). When the available height is divided up, the proportion of the available height given to each row is determined by dividing the row heights by the sum of the row heights. For example, in the code `layout(matrix(c(1, 2)), height = c(2,1))` the top row is given two-thirds of the available height (2/(2+1)) and the bottom row is given one third (1/(2+1)).

```
> (m <- matrix(c(1, 2)))

     [,1]
[1,]    1
[2,]    2

> layout(m, height = c(2, 1))
> layout.show(2)
```



By default, the division of row heights is completely independent of the division of column widths (`respect = FALSE`). Use `respect = TRUE` to force the widths and heights to correspond as well so that, for example, a height of `1` corresponds to the same physical distance as a width of `1`. The `respect=` argument can also be specified as a matrix. In this case, only certain rows and columns will respect each other's heights/widths. As an example, let's look at the relationship between age in years and serum albumin, but also include each variable's histogram – see Figure 4.16.

OVERLAYING OUTPUT: As we know, all high-level plotting functions always start a new plot, erasing the current plot if necessary. However, we may want to generate a plot that results from the superimposing of two high-level plotting functions. Some high-level functions provide an argument called `add=`, which, if set to `TRUE`, will add the function output to the current plot, rather than starting a new plot. We have demonstrated this in the first document. Unfortunately, the `add=` argument is only available in some high-level plotting functions. Alternatives include using the `new=` parameter via the `par()` function, or the `plot.new()` function. When `new = TRUE`, the next high-level plotting function will simply overlay the existing plot. ***BE AWARE***, unlike the `add=` argument, using the `new= par()` function parameter or the `plot.new()` function *does not* guarantee that the axes from your two plots will line up exactly. In actuality, the axes will often conflict with each other. To make sure they don't, explicitly define the ranges of both axes in both high-level function invocations using the `xlim=` and `ylim=` arguments.

OTHER ADDITIONS: As we demonstrated in the first document, the traditional graphics system provides the `legend()` function for adding a legend or key to a plot. The legend is usually drawn within the plot region, and is located relative to user coordinates. The function has many arguments, which allow for a great deal of flexibility in the specification of the contents and the layout of the legend. A legend can also been drawn outside the plot region (ie, in a margin) by specifying `xpd = NA` using the `par()` function – see the 'Plotting regions & Margins' section. In turn, the location coordinates of the legend specified in the subsequent `legend()` function invocation should be outside the plotting region. It should be noted that it is entirely your responsibility to ensure that the legend corresponds to the plot. There is no automatic checking that data symbols in the legend match those in the plot, or that labels in the legend have any correspondence with the data.

The `locator()` function is a very useful one to know about and allows you to interactively select the position for graphical elements such as legends or labels. Specifically, the `locator()` function waits for you to select locations on the current plot using the left mouse button. This continues until `n` (default 512) points have been selected, or another mouse button is pressed. As an example, we could use the following code to place some informative text near an outlying point: `text(locator(1), "Outlier", adj = 0)`. The `locator()` function will be ignored if the current device, such as `postscript` does not support interactive pointing.

Another useful function is the `rug()` function, which produces a 'rug' plot along one of the axes of your existing plots. A 'rug' plot consists of a series of tick marks that represent data locations. This can be useful to represent an additional one-dimensional plot of your data (e.g., in combination with a density curve). For example, let's add a rug plot to a histogram of a random normal variable – see Figure 4.17. Alternatives to the `rug()` function are the `Hmisc` package's `scat1d()`, `datadensity()`, and `histSpike()` functions, which adds tick marks corresponding to non-missing values of the data on any of the four sides of an existing plot, a graphical representation of data density, and a high-resolution data distribution that is particularly good for very large datasets (say N > 1000), respectively.

MATHEMATICAL ANNOTATION: Any high- or low-level graphics function or graphical function's argument that draws text should accept both a normal text string (e.g., `"some text"`), and an R expression, which is typically the result of a call to the `expression()` function. If an expression is specified as the text to draw, then it is interpreted as a mathematical annotation and is formatted appropriately. For a complete description of the available mathematical annotations, invoke `demo(plotmath)`, which steps you through several tables. In these tables, the columns of gray text show sample R expressions, and the columns of black text show the resulting output. As an example (Figure 4.18), let's generate a blank plot and then add various mathematical annotations as text to the plot – notice the use of 'double equals' (`==`) to print a single

Figure 4.16: Creating multiple plots on one page using the `layout()` function.

```
> library(Hmisc)
> op <- par(no.readonly = TRUE)
> layout(matrix(c(1, 0, 3, 2), ncol = 2, nrow = 2, byrow = TRUE), widths = c(3,
+     1), heights = c(1, 3), respect = TRUE)
> agehist <- with(pbc, hist(ageyrs, plot = FALSE))
> albumhist <- with(pbc, hist(album, plot = FALSE))
> top <- max(agehist$counts, albumhist$counts)
> par(mar = c(0, 3, 1, 1))
> barplot(albumhist$counts, axes = FALSE, ylim = c(0, top), space = 0)
> par(mar = c(3, 0, 1, 1))
> barplot(agehist$counts, axes = FALSE, xlim = c(0, top), space = 0, horiz = TRUE)
> par(mar = c(4, 4, 1, 1))
> with(pbc, plot(ageyrs ~ album, xlab = paste(label(album), " (", units(album),
+     ")", sep = ""), ylab = paste(label(ageyrs), " (", units(ageyrs),
+     ")", sep = "")))
> par(op)
```

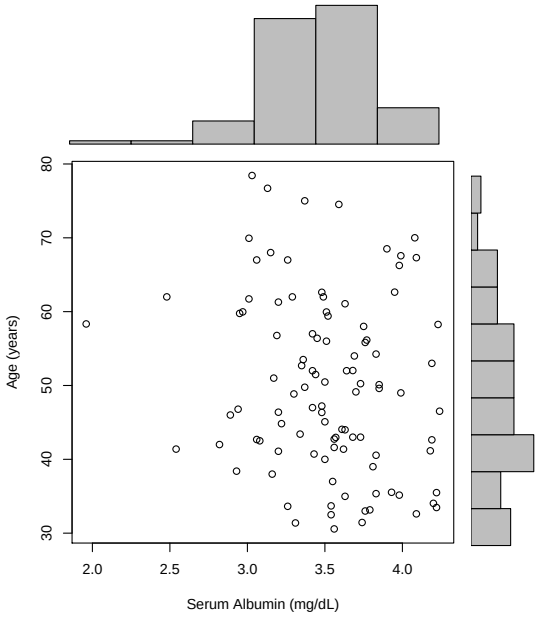
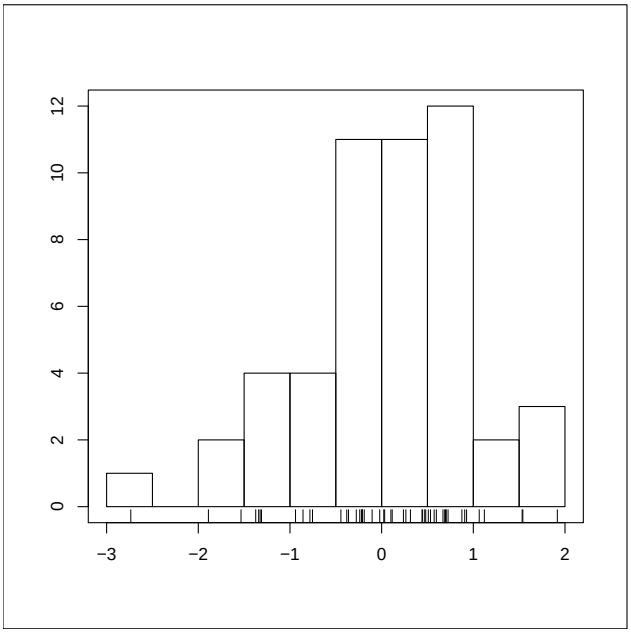


Figure 4.17: Adding a 'rug' plot to an existing plot using the low-level `rug()` function.

```
> y <- rnorm(50)
> hist(y, main = "", xlab = "", ylab = "")
> box()
> rug(y)
> box("figure")
```



equal sign. We can also incorporate the `paste()` function into our expression – see Figure 4.19. Often, we will want to replace some variable in a mathematical annotation with its values. To do this, we need to use the `substitute()` function instead of the `expression()` function. We must specify the substitution using a *list* as a second argument and the value to be substituted must have previously been computed and assigned a name. See Figure 4.20 for an example.

GRAPHICAL OUTPUT: As demonstrated in the first document, when using R interactively, the main persistent record of graphical output is a window on your screen, which is also known as a GUI ('graphical user interface') screen *device*. When R is installed, an appropriate screen format is selected as the default screen device and this default device is opened automatically the first time any graphical output occurs. For example, on the various Unix systems, the default GUI screen device is an X11 window. These GUI screen devices are opened by internally calling the `x11()`, `windows()`, and `quartz()` functions in the Linux/Unix, Windows, and Mac versions of R, respectively. By default, the size of an X11 graphics window and a Windows graphics window are 7 inches by 7 inches. A Mac graphics window is, by default, 5 inches by 5 inches. The three mentioned functions all have `height=` and `width=` arguments that can be used to open new screen devices of the desired size.

It is possible to have more than one screen device open at the same time, but only one device is currently 'active' and all graphics output is sent to that device. If multiple devices are open, there are functions to control which device is active. The screen devices are associated with a name (e.g., `"X11"` or `"postscript"`) and a number – the `"null device"` is always device 1. The list of open devices can be obtained using the `dev.list()` function, which returns the name and number of all open devices, except device 1, the null device. The `dev.cur()` function returns the number and name of only the active device, or 1, the null device, if none is active. The `dev.set()` function can be used to make a device active by specifying the appropriate device number. And the `dev.next()` and `dev.prev()` functions can be used to make the next/previous device on the device list the active device. *Be aware*, the display list can consume a reasonable amount of memory if a plot is particularly complex or if there are very many devices open at the same time. The `dev.off()` shuts down the specified (by default the current) device, and the `graphics.off()` functions shuts down all open graphics devices.

It is also possible to *partition* the active screen device with the `split.screen()` function. For example: `split.screen(c(1, 2))` divides the device into two parts which can be selected with `screen(1)` or `screen(2)`. A part of the partitioned screen device can itself be divided again with the `split.screen()` function, which can make complex partitions. The `erase.screen()` function clears a single screen, and the `close.screen()` removes the specified screen partition. These functions are totally incompatible with the other mechanisms for arranging plots on a device (i.e., the `par()` function's `mfrow=` argument and the `layout()` function), and some plotting functions like the `coplot()` function.

Also, in the Windows version of R, you can save a 'history' of your graphs by activating the Recording feature under the History drop-menu (seen when the graphics window is selected). You can then access old graphs by using the Page Up and Page Down keys.

As we mentioned in the first document, it is possible to produce a *file* that contains your plot. Similar to the screen devices, the graphical output can be directed to a particular *file device*, which dictates the output format that will be produced. And like the screen devices, the file devices are controlled by specific functions: the `postscript()` function produces an Adobe PostScript file; the `pdf()` function produces an Adobe PDF file; the `pictex()` function produces a LaTeX PicTEX file; the `xfig()` function produces an XFIG file; the `bitmap()` function produces a GhostScript conversion to a file; the `png()` function produces a PNG bitmap file; and the `jpeg()` function produces a JPEG bitmap file. On the Windows version of R, there are also the `win.metafile()` and `bmp()` file device functions, which produce a Windows Metafile file and Windows BMP file, respectively. Like the screen device functions, these file device functions allow you to specify things such as the name of the file and the size of the plot.

Figure 4.18: Examples of mathematical annotation in a graph using the `expression()` function.

```
> par(mar = c(1, 1, 1, 1))
> plot(0:10, 0:10, type = "n", axes = FALSE)
> text(1, 10, expression(x %+-% y), cex = 1.5)
> text(1, 9, expression(x[i]), cex = 1.5)
> text(1, 8, expression(x^2), cex = 1.5)
> text(1, 7, expression(sqrt(x)), cex = 1.5)
> text(1, 6, expression(sqrt(x, 3)), cex = 1.5)
> text(1, 5, expression(x != y), cex = 1.5)
> text(1, 4, expression(x <= y), cex = 1.5)
> text(1, 3, expression(hat(x)), cex = 1.5)
> text(1, 2, expression(tilde(x)), cex = 1.5)
> text(1, 1, expression(bar(x)), cex = 1.5)
> text(1, 0, expression(x %<=>% y), cex = 1.5)
> text(4, 10, expression(Alpha + Omega), cex = 1.5)
> text(4, 9, expression(alpha + omega), cex = 1.5)
> text(4, 8, expression(45 * degree), cex = 1.5)
> text(4, 7, expression(frac(x, y)), cex = 1.5)
> text(4, 5.5, expression(sum(x[i], i = 1, n)), cex = 1.5)
> text(4, 4, expression(prod(plain(P)(X == x), x)), cex = 1.5)
> text(4, 2.5, expression(integral(f(x) * dx, a, b)), cex = 1.5)
> text(4, 0.5, expression(lim(f(x), x %->% 0)), cex = 1.5)
> text(8, 10, expression(x^y + z), cex = 1.5)
> text(8, 9, expression(x^(y + z)), cex = 1.5)
> text(8, 8, expression(x^{
+     y + z
+ }), cex = 1.5)
> text(8, 6, expression(hat(beta) == (X^t * X)^{
+     -1
+ } * X^t * y), cex = 1.5)
> text(8, 4, expression(bar(x) == sum(frac(x[i], n), i == 1, n)), cex = 1.5)
> text(8, 2, expression(paste(frac(1, sigma * sqrt(2 * pi)), " ", plain(e)^{
+     frac(-(x - mu)^2, 2 * sigma^2)
+ })), cex = 1.5)
> box("figure")
```

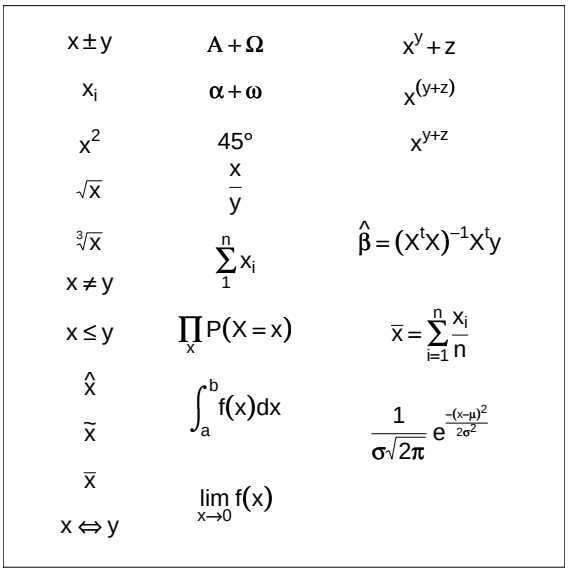


Figure 4.19: Examples of mathematical annotation in a graph using the `paste()` function in conjunction with the `expression()` function.

```
> x <- seq(-4, 4, length = 101)
> plot(x, sin(x), type = "l", xaxt = "n", xlab = expression(paste("Phase Angle ",
+     phi)), ylab = expression("sin " * phi))
> axis(side = 1, at = c(-pi, -pi/2, 0, pi/2, pi), label = expression(-pi,
+     -pi/2, 0, pi/2, pi))
> box("figure")
```

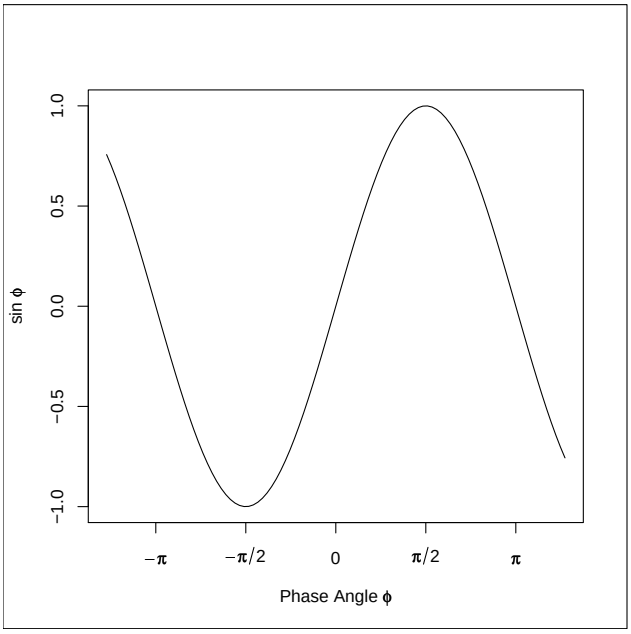
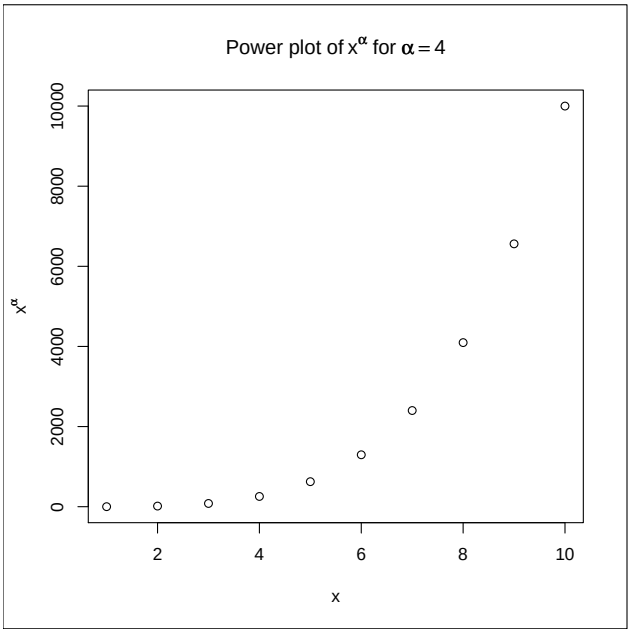


Figure 4.20: Examples of mathematical annotation in a graph using the `substitute()` function.

```
> x <- 1:10
> alpha <- 4
> plot(x, x^alpha, xlab = "x", ylab = expression(x^alpha), main = substitute(paste("Power plot of ",
+     x^alpha, " for ", alpha == ch.a), list(ch.a = alpha)))
> box("figure")
```



Unlike sending graphical output to a window on your screen, directing graphical output to a file takes a few more steps. A file device must be created or 'opened to writing' in order to receive graphical output by invoking the desired file device function with at least the desired file name specified. Once you have opened the file to writing, you then compile all of your desired graphical function invocations. In turn, the specific file device that you opened converts the graphical function invocations from R (e.g., 'draw a line') into commands that the particular device can understand (e.g., PostScript commands if you used the `postscript()` function). When you have finished writing the desired graphical output to a file, you then close the file to writing (and therefore close the file device) by invoking the `dev.off()` function. For example, let's write a simple scatterplot to a PDF file named `myplot.pdf`:

```
> pdf("myplot.pdf", height = 8.5, width = 11)
> with(pbc, plot(age.years ~ chol, main = "Age (years) vs. Serum Chol",
+    xlab = label(chol), ylab = label(age.years)))
> dev.off()
```

In the previous code, we also specified that the plot should be in a landscape orientation (i.e., height $<$ width), with a height of 8.5" and a width of 11.0".

Because all graphical output is directed to a file when using a file device function, it is best to use the appropriate screen device function to specify the size of the graphics window on your screen, and to direct all intermediate iterations of the graphical output to the window. Once the graph has been 'finalized,' you can then use one of the file device functions to write the graph to a file.

Also, when using the Windows version of R, you do not necessarily need to save the graph as its own file using a file device function. Specifically, in the Windows version of R, right-clicking on the graphics window offers you three options for outputting any desired graph from R: (1) Copy your graph as either a metafile or a bitmap; (2) Save your graph as either a metafile or postscript; and/or (3) Print your graph. For the Copy and Save options, it is very easy to then Paste or Insert, respectively, your graph into a Microsoft Word and/or Powerpoint document.

For a screen device, starting a new page involves clearing the window before producing more output. On the other hand, only certain file devices allow multiple pages of output. For example, PostScript and PDF allow multiple pages, but PNG does not. It is usually possible, especially for devices that do not support multiple pages of output, to specify that each page of output produces a separate file. This is achieved by specifying the argument `onefile = FALSE` when opening a device and specifying a pattern for the file name like `file = "myplot%03d"`. The `%03d` is replaced by a three-digit number (padded with zeros) indicating the 'page number' for each file that is created.