

First Steps with *R*

Seth Falcon

Martin Morgan

July 29, 2010

Contents

1	Install <i>R</i>	2
2	Atomic Vectors – <i>R</i>’s Basic Types	2
3	Subsetting Vectors	7
3.1	Integer subsetting	8
3.2	Logical subsetting	9
3.3	Subsetting by name	10
4	Matrix, <i>data.frame</i>, and 2-D Subsetting	11
5	Lists	14
5.0.1	Extracting elements from a list	16
5.0.2	Subsetting lists	16
6	Environments	18
7	Functions	20
8	Getting Help in <i>R</i>	21
8.1	HTML Help System	21
8.2	Vignettes	22
9	Exploring <i>R</i> objects	22
10	Session information	23

1 Install *R*

The first step is to select a nearby CRAN mirror site from the list at <http://cran.r-project.org/mirrors.html>. You should now have a web browser open displaying the main page of your selected CRAN mirror.

Now follow the instructions below appropriate for your operating system:

Windows 1. Click the *Windows* link.

2. Click the *base* link for “Binaries for base distribution (managed by Duncan Murdoch)” and then download the main link for *Download R 2.11.1 for Windows*.
3. Double-click the file you downloaded (*R-2.11.1-win32.exe*) and follow the installation prompts. You can accept all of the default values.
4. Start *R* by double-clicking on the *R* icon on your desktop or selecting the entry from the Start, Program Files menu.

Mac OS X 1. Click the *MacOS X* link.

2. Download the first link with name *R-2.11.1.pkg*. This is a universal binary for OS X 10.5 and higher. If you have an older version of OS X, read the details on that page to determine which package to download.
3. Double-click the downloaded pkg file and follow the installation prompts. You can accept all of the default values.
4. Start *R* by clicking the *R.app* icon in your Applications folder.

Linux

- Your Linux distribution’s package manager may already have the latest version of *R* available (*R-2.11.1*). If not there may be a precompiled binary package available. Click the *Linux* link on CRAN and follow the instructions based on your distribution.
- You can build from source instead of using a precompiled binary. In this case, download the source package from the main page of your CRAN mirror: *R-2.11.1.tar.gz*. Installation instructions can be found here: <http://cran.fhcrc.org/doc/manuals/R-admin.html>
- If *R* is not in your *PATH*, add it. Then start *R* with the *R* command.

2 Atomic Vectors – *R*’s Basic Types

There are six basic data types in *R* that are always represented as *vectors* (see Table 2). A vector is a one-dimensional array of items of the same type. In *R*, these vectors are called *atomic vectors* because they cannot be split into individual items; single items are represented as vectors of length one.

type	example
logical	TRUE, FALSE
integer	0L, 1L
numeric	1, 3.14
complex	1+1i
character	"a"
raw	as.raw(255)

Table 1: The atomic vector data types in *R*. The “L” suffix indicates an integer literal in *R*code; digits with no decimal point are interpreted as type *numeric* (real).

Since “everything’s a vector”¹, most functions in *R* are *vectorized*. Vectorized functions accept vectors as input and perform an action element-wise on the input. Consider the following example:

```
> ## length, width, and height measurements
> L <- c(1.2, 4.3, 2.3, 3)
> W <- c(13.8, 22.4, 18.1, 17)
> H <- c(7, 7, 10, 3.4)
> volume <- L * W * H
> volume

[1] 115.92 674.24 416.30 173.40

> total_length <- sum(L)
> total_length

[1] 10.8
```

The example above defines three *numeric* vectors representing length, width, and height measurements of four objects. Vector literals are constructed using the `c` function which should be given a comma separated list of items of the same type to concatenate into a vector. The assignment operator in *R* is `<-` although you can also use `=`. Comments begin with `#` and continue to the end of the line. The volume of each object is computed by vectorized multiplication using `*` which operates element-wise. The `sum` function accepts a vector and returns a new vector of length one containing the sum of the elements of the input.

Start up a new *R* session and try the following exercises:

Exercise 1

Create a vector representing the radii of three circles with lengths 5, 10, and 20. Use `*` and the built-in constant `pi` to compute the areas of the three circles. Then subtract 2.1 from each radius and recompute the areas.

¹Well, almost everything. *R* also has *lists*, *environments*, *S3*, and *S4* classes none of which are vectors

Solution:

```

> circles <- c(5, 10, 20)
> # areas are:
> pi * circles * circles

[1] 78.53982 314.15927 1256.63706

> # you could also use ^
> pi * circles^2

[1] 78.53982 314.15927 1256.63706

> # reduce radii by 2.1
> pi * (circles - 2.1)^2

[1] 26.42079 196.06680 1006.59770

```

Exercise 2

Creating regular numeric sequences is a common task in statistical computing. You can use the `seq` function to create sequences as well as a shorthand : (e.g. `1:10`).

1. Read the help page for `seq` by entering `help(seq)`.
2. Generate a decreasing sequence from 50 to 1, then another sequence from 1 to 50. See if you can understand the meaning of the `[i]` prefix that R uses to print vectors.
3. Use `seq` to generate a sequence of the even integers between one and ten.

Solution:

```

> ## sequences from 100 to 1 and 1 to 100
> 50:1 # or seq(100, 1)

[1] 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33
[19] 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15
[37] 14 13 12 11 10 9 8 7 6 5 4 3 2 1

> 1:50 # or seq(1, 100)

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50

```

```

> ## the [i] prefix tells you the index of the first
> ## item on the current line within the vector
> ## being displayed.
>
> ## even integers
> seq(2, 10, 2)

[1] 2 4 6 8 10

```

Exercise 3

All R functions have a manual page that you can access via `help(funcName)`, or the shorthand `?funcName`. Most manual pages include an example section which you can execute in your session by calling `example(funcName)`. Run the examples for the `paste` function. Use `paste` to create a character vector of length one that looks like "id-1, id-2, id-3".

Solution:

```

> example(paste)

paste> x <- Rle(10:1, 1:10)

paste> x
'integer' Rle of length 55 with 10 runs
Lengths: 1 2 3 4 5 6 7 8 9 10
Values : 10 9 8 7 6 5 4 3 2 1

paste> runLength(x)
[1] 1 2 3 4 5 6 7 8 9 10

paste> runValue(x)
[1] 10 9 8 7 6 5 4 3 2 1

paste> nrun(x)
[1] 10

paste> diff(x)
'integer' Rle of length 54 with 18 runs
Lengths: 1 1 1 2 1 3 1 4 ... 6 1 7 1 8 1 9
Values : -1 0 -1 0 -1 0 -1 0 ... 0 -1 0 -1 0 -1 0

paste> unique(x)
[1] 10 9 8 7 6 5 4 3 2 1

paste> sort(x)
'integer' Rle of length 55 with 10 runs

```

```

Lengths: 10  9  8  7  6  5  4  3  2  1
Values :  1  2  3  4  5  6  7  8  9 10

paste> sqrt(x)
'numeric' Rle of length 55 with 10 runs
Lengths:          1 ...          10
Values : 3.16227766016838 ...      1

paste> x^2 + 2 * x + 1
'numeric' Rle of length 55 with 10 runs
Lengths:  1  2  3  4  5  6  7  8  9 10
Values : 121 100  81  64 49 36 25 16  9  4

paste> x[c(1,3,5,7,9)]
'integer' Rle of length 5 with 4 runs
Lengths:  1  1  1  2
Values : 10  9  8  7

paste> window(x, 4, 14)
'integer' Rle of length 11 with 3 runs
Lengths: 3 4 4
Values : 8 7 6

paste> range(x)
[1]  1 10

paste> table(x)

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

paste> sum(x)
[1] 220

paste> mean(x)
[1] 4

paste> x > 4
'logical' Rle of length 55 with 2 runs
Lengths:  21  34
Values : TRUE FALSE

paste> aggregate(x, x > 4, mean)
[1] 6.666667 2.352941

paste> aggregate(x, FUN = mean, start = 1:(length(x) - 50), end = 51:length(x))

```

```
[1] 4.235294 4.058824 3.901961 3.745098 3.607843
```

```
paste> y <- Rle(c(TRUE,TRUE,FALSE,FALSE,TRUE,FALSE,TRUE,TRUE,TRUE))
```

```
paste> y
'logical' Rle of length 9 with 5 runs
Lengths:  2    2    1    1    3
Values : TRUE FALSE TRUE FALSE TRUE
```

```
paste> as.vector(y)
[1] TRUE TRUE FALSE FALSE TRUE FALSE TRUE TRUE TRUE
```

```
paste> rep(y, 10)
'logical' Rle of length 90 with 41 runs
Lengths:  2    2    1    1 ...    1    1    3
Values : TRUE FALSE TRUE FALSE ... TRUE FALSE TRUE
```

```
paste> c(y, x > 5)
'logical' Rle of length 64 with 6 runs
Lengths:  2    2    1    1   18   40
Values : TRUE FALSE TRUE FALSE TRUE FALSE
```

```
paste> z <- c("the", "quick", "red", "fox", "jumps", "over", "the", "lazy", "brown", "dog")
```

```
paste> z <- Rle(z, seq_len(length(z)))
```

```
paste> chartr("a", "@", z)
'character' Rle of length 55 with 10 runs
Lengths:  1    2    3 ...    9    10
Values :  "the" "quick" "red" ... "brown" "dog"
```

```
paste> toupper(z)
'character' Rle of length 55 with 10 runs
Lengths:  1    2    3 ...    9    10
Values :  "THE" "QUICK" "RED" ... "BROWN" "DOG"
```

```
> paste("id", 1:3, sep="-", collapse=" ")
```

```
[1] "id-1, id-2, id-3"
```

3 Subsetting Vectors

Next we will take a tour of the ways that one can slice, extract, index, subset and otherwise get data out of vectors in *R*. We will use the following vector *v* as an example.

```

> ## You can name the elements of a vector
> ## The names do not have to be unique, but
> ## often you will want them to be.
> v <- c(a = 1.1, b = 2, b = 100, c = 50, d = 60)
> v

      a      b      b      c      d
1.1    2.0 100.0  50.0  60.0

```

3.1 Integer subsetting

```

> v[1]

a
1.1

> v[length(v)]

d
60

> v[3:5]

      b      c      d
100    50    60

> v[c(1, 1, 4, 4)]

      a      a      c      c
1.1    1.1 50.0  50.0

```

Exercise 4

Create an integer vector *i* that can be used to subset *v* such that it will output the elements of *v* in decreasing order. For the general case, read the help pages for *order* and *sort*.

Solution:

```

> ## you need the L's if you want an _integer_ vector,
> ## otherwise you get a numeric (double) vector.
> v[c(3L, 5L, 4L, 2L, 1L)]

      b      d      c      b      a
100.0  60.0  50.0   2.0   1.1

> v[order(v, decreasing = TRUE)]

      b      d      c      b      a
100.0  60.0  50.0   2.0   1.1

```


Zero values are ignored in integer subsetting and negative values can be used to exclude elements (positive and negative indices cannot be mixed).

```
> v[c(1, 0, 0, 0, 0, 3)]

      a      b
1.1 100.0

> v[-2]

      a      b      c      d
1.1 100.0  50.0  60.0

> v[seq(1, length(v), 2)]

      a      b      d
1.1 100.0  60.0

> v[-seq(1, length(v), 2)] # note the '-' sign

      b      c
      2 50

> ## two other special cases
> v[]

      a      b      b      c      d
1.1   2.0 100.0  50.0  60.0

> v[integer(0)]

named numeric(0)
```

3.2 Logical subsetting

In a subset expression `v[i]` when `i` is a logical vector (TRUE/FALSE) with the same length as `v` we refer to it as *logical subsetting*. Values of `v` that align with a TRUE in `i` are selected.

```
> v[c(TRUE, FALSE, FALSE, FALSE, FALSE)]

      a
1.1

> v > 5

      a      b      b      c      d
FALSE FALSE  TRUE  TRUE  TRUE

> v[v > 5]
```

```

      b    c    d
100  50   60

> v[!(v > 5)] # or (v <= 5)

      a    b
1.1 2.0

> v[(v > 5) & (v < 90)] # element-wise AND

      c    d
50 60

> v[(v < 5) | (v > 90)] # element-wise OR

      a      b      b
1.1    2.0 100.0

> v[v == 100]

      b
100

```

Exercise 5

The names of a vector can be obtained using *names*. Create a logical vector that has a *TRUE* for all names equal to “b”. Use this logical vector to extract all elements of *v* with a name of “b”.

Solution:

```

> v[names(v) == "b"]

      b    b
2 100

```

3.3 Subsetting by name

Finally, in the case of a named vector, you can use the names to select elements. If the vector has non-unique names, the first element to match is returned.

```

> v["c"]

      c
50

> v[c("d", "a")]

      d      a
60.0  1.1

```

```
> v["b"] # only get first match
```

```
b  
2
```

Exercise 6

Use `sample` to randomly select three names of `v` and then use the result to extract the corresponding elements of `v`.

Solution:

```
> set.seed(5644L)  
> i <- sample(names(v), 3)  
> v[i]
```

```
  c  d  b  
50 60  2
```

4 Matrix, *data.frame*, and 2-D Subsetting

After vectors, two very common data structures in *R* are the *matrix* and the *data.frame*. A *matrix* is a two-dimensional vector. Like vectors, all elements share a common type. The *data.frame* class groups together a set of vectors (columns) of the same length. A *data.frame* is analogous to a table in a relational database, and is used to represent statistical data sets where each row represents a sample and each column an attribute of the sample. The columns can be of different types, but within a column all elements are the same type.

Subsetting a matrix or *data.frame* is similar to subsetting vectors, except that two arguments are given to the `[]` operator. All of the subsetting approaches (integer, logical, character) that you've used with vectors can be used with matrices and *data.frames*. Consider the following examples and then explore the exercises that follow.

```
> m <- matrix(1:25, ncol = 5,  
+           dimnames = list(letters[1:5], LETTERS[1:5]))  
> m
```

```
  A  B  C  D  E  
a 1  6 11 16 21  
b 2  7 12 17 22  
c 3  8 13 18 23  
d 4  9 14 19 24  
e 5 10 15 20 25
```

```
> ## extract an element  
> m[3, 1]
```

```

[1] 3

> ## subset the matrix
> m[1:3, 1:3]

  A B C
a 1 6 11
b 2 7 12
c 3 8 13

> ## empty index argument means
> ## select all of the dimension
> m[ , 2] # 2nd column all rows

  a  b  c  d  e
6  7  8  9 10

> m[4, ] # 4th row all columns

  A  B  C  D  E
4  9 14 19 24

```

Exercise 7

Create a new matrix from *m* by removing the second row and the fourth column.
Hint: use negative indices.

Solution:

```

> m[-2, -4]

  A  B  C  E
a 1  6 11 21
c 3  8 13 23
d 4  9 14 24
e 5 10 15 25

```

Exercise 8

Subset *m* such that you keep only rows where the value in the “D” column is greater than 17.

Solution:

```

> d_gt17 <- m[ , "D"] > 17
> m[d_gt17, ]

  A  B  C  D  E
c 3  8 13 18 23
d 4  9 14 19 24
e 5 10 15 20 25

```

Exercise 9

Find the element-wise product of rows “b” and “d”.

Solution:

```
> m["b", ] * m["d", ]

  A   B   C   D   E
8  63 168 323 528
```

We'll use the `Indometh` data set that comes with *R* to work some examples with the `data.frame` class. If you want to know what this data represents, call `help(Indometh)`.

```
> ## load a dataset that comes with R
> data(Indometh)
> df <- Indometh # use a shorter name
> class(df)

[1] "nfnGroupedData" "nfGroupedData"  "groupedData"
[4] "data.frame"

> dim(df)

[1] 66  3

> names(df)

[1] "Subject" "time"    "conc"

> df[1:3, ]

  Subject time conc
1        1 0.25 1.50
2        1 0.50 0.94
3        1 0.75 0.78

> ## The following are all ways of
> ## extracting the time column
> ##
> ## df[, "time"]; df[, 2]
> ## df[["time"]]; df[[2]]
> ## df$time
> identical(df[, 2], df$time)

[1] TRUE
```

Exercise 10

1. Extract the rows of `df` that contain data for subject 2.

2. Extract the rows for time point 0.50.
3. Which subjects had a concentration greater than 0.38 at time point 2.0?

Solution:

```
> ## Subject 2
> head(df[df$Subject == "2", ])

  Subject time conc
12      2 0.25 2.03
13      2 0.50 1.63
14      2 0.75 0.71
15      2 1.00 0.70
16      2 1.25 0.64
17      2 2.00 0.36

> ## time point 0.50
> head(df[df$time == 0.50, ])

  Subject time conc
2        1  0.5 0.94
13       2  0.5 1.63
24       3  0.5 1.49
35       4  0.5 1.39
46       5  0.5 1.04
57       6  0.5 1.44

> ## conc > 0.38 at time point 2.0
> df[df$time == 2.0 & df$conc > 0.38, ]

  Subject time conc
28       3     2 0.39
39       4     2 0.40
61       6     2 0.42
```

5 Lists

A limitation of *R*'s atomic vectors is that all items must be of the same type. *R* provides two data types for organizing arbitrary collections of *R* objects: *list* and *environment*. Both are *recursive* data structures meaning that a list can contain other lists (as well as other types) and an environment can contain other environments (and other types). Lists are discussed below. See Section 6 for an introduction to *environment*.

In Section 4 you worked with a *data.frame* which at its core is a *list* of vectors. In fact, except for the 2-dimensional subsetting, you will see many similarities between *list* subsetting and accessing the columns of a *data.frame*²

```
> things <- list(a = 1:3, b = c("X", "Y"),
+               uspaper = list(length = 11, width = 8.5, units = "in"),
+               eupaper = list(length = 297, width = 210, units = "mm"),
+               TRUE)
> things

$a
[1] 1 2 3

$b
[1] "X" "Y"

$uspaper
$uspaper$length
[1] 11

$uspaper$width
[1] 8.5

$uspaper$units
[1] "in"

$eupaper
$eupaper$length
[1] 297

$eupaper$width
[1] 210

$eupaper$units
[1] "mm"

[[5]]
[1] TRUE

> names(things)

[1] "a"      "b"      "uspaper" "eupaper" ""

> length(things)
```

²In many cases, the underlying code is the same: a *data.frame* is a *list*.

```
[1] 5
```

List elements, like items in a vector, can be named, but this is not required. In the example above, all elements of `things` are named except for the last element, a logical vector of length one containing the value `TRUE`.

You can extract the element names of a list using `names` just as you can with a vector (recall that for a *data.frame*, `names` returns the names of the columns).

5.0.1 Extracting elements from a list

You can extract *elements* of a list using double square brackets as shown below.

```
> ## by position using [[
> things[[4]]
```

```
$length
[1] 297
```

```
$width
[1] 210
```

```
$units
[1] "mm"
```

```
> ## by name using [
> things[["b"]]
```

```
[1] "X" "Y"
```

```
> ## by name using $
> things$a
```

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

5.0.2 Subsetting lists

List subsetting is achieved using single square brackets which will always return a new list.

```
> ## by index
> things[2]
```

```
$b
[1] "X" "Y"
```

```
> things[c(1, 5)]
```



```

$a
[1] 1 2 3

[[2]]
[1] TRUE

> ## negative indices remove just like vectors
> things[-c(3, 4)]

$a
[1] 1 2 3

$b
[1] "X" "Y"

[[3]]
[1] TRUE

> ## by name
> things[c("a", "b")]

$a
[1] 1 2 3

$b
[1] "X" "Y"

> ## logical works too
> hasLenTwo <- sapply(things, function(x) length(x) == 2)
> hasLenTwo

      a      b uspaper eupaper
FALSE  TRUE  FALSE  FALSE  FALSE

> things[hasLenTwo]

$b
[1] "X" "Y"

```

Exercise 11

Explain the difference between `things[["uspaper"]]` and `things["uspaper"]`.

Solution: Using double square brackets extracts an element from the `things` list, in this case a list with three elements. Single square brackets subsets the `things` list and returns a list of length one, its first and only element is the list of length three returned by the double bracket expression.

Exercise 12

Extract the “uspaper” and “eupaper” lists from `things` and assign each to a variable. Subset each list to remove the element with name “units”. Convert each resulting two-element list to a numeric vector using `as.numeric`. Convert the uspaper values to millimeters by multiplying by 25.4. Finally, use `prod` to compute the areas of the two paper sizes. Can you compute each area in a one-liner without assigning temporary variables?

Solution:

```
> usp <- things[["uspaper"]]
> usp <- as.numeric(usp[-3])
> usp <- 25.4 * as.numeric(usp)
> eup <- things[["eupaper"]]
> eup <- as.numeric(eup[-3])
> prod(usp)

[1] 60322.46

> prod(eup)

[1] 62370

> ## one line versions
> prod(25.4 * as.numeric(things[["uspaper"]][-3]))

[1] 60322.46

> prod(as.numeric(things[["eupaper"]][-3]))

[1] 62370
```

6 Environments

The *environment* type is a data structure that maps keys to values. In other programming languages these structures may be called dictionaries, hash tables, or associative arrays. The keys of R’s *environments* must be strings, but the values can be arbitrary R objects. The items in an *environment* are not ordered (unlike a named list).

Here is what basic *environment* assignment and extraction looks like:

```
> env <- new.env(parent = emptyenv())
> ## we haven't talked about assignment, but similar
> ## forms also work for lists (and with "[" for vectors).
> env[["a"]] <- 1:3
> env$b <- "hello"
> env[["b"]]
```

```

[1] "hello"
> env$a
[1] 1 2 3
> mget(c("b", "a", "b"), env)
$b
[1] "hello"

$a
[1] 1 2 3

$b
[1] "hello"
> ## list all keys
> ls(env)
[1] "a" "b"

```

The *environment* data type behaves differently than (almost) all other data types in R. Other data types in R have pass-by-value semantics, which means that when you pass an object to a function or assign it to a new variable name you make a *copy* of the original object. *environments* have pass-by-reference semantics, which means that *no copy* is made when you pass an *environment* to a function or assign it to a new variable name. Here's an example to explore:

```

> ## create an environment
> env1 <- new.env(parent = emptyenv())
> env1[["name"]] <- "alice"
> env1[["count"]] <- 100
> ## create a similar list
> lst1 <- list(name = "bob", count = 100)
> ## this just creates a new reference,
> ## NOT a copy
> env2 <- env1
> ## this effectively copies the list
> lst2 <- lst1
> ## now modify original environment and list
> env1[["count"]] <- 200
> lst1[["count"]] <- 200
> ## env2 points to same data as env1
> env2[["count"]]

[1] 200

> ## but lst2 is a copy and was not changed
> lst2[["count"]]

[1] 100

```

7 Functions

Defining your own functions is an essential part of creating efficient and reproducible analyses as it allows you to organize a sequence of computations into a unit that can be applied to any set of appropriate inputs.

Here's a basic function definition:

```
> say <- function(name, greeting = "hello")
+ {
+   paste(greeting, name)
+ }
> ## these two calls use positional argument matching
> say("world")

[1] "hello world"

> say("world", "goodbye")

[1] "goodbye world"

> ## this call matches arguments by name, order
> ## doesn't matter for this case
> say(greeting = "g'day", name = "Seattle")

[1] "g'day Seattle"
```

The *name* of this function is `say_hello`. It has one *formal argument*: `name`. The *body* of the function is between the curly braces. The return value of a function in *R* is the value of the last evaluated expression in the body. Arguments to a function can specify default values, as is the case with `greeting` above. The default value is used if a value is not provided when the function is called.

When calling functions in *R*, you can provide arguments in the same order as in the definition of the function, or you can name the arguments as shown in the last call to `say` above. Naming arguments is a good practice because it makes code more self-explanatory and robust (a change in the function's argument order won't impact your call, for example).

Exercise 13

Write a function that separates the data in the `Indometh` `data.frame` by subject. Your function should take a single argument and return a list of `data.frames` such that each `data.frame` has the data for one of the subjects. The returned list should have names `Subject1`, `Subject2`, ..., `Subject6` when given the `Indometh` `data.frame` as input. Also, since the `Subject` column is now redundant, remove it from the subject-specific `data.frames` returned by your function.

Solution:

```
> splitBySubject <- function(df)
+ {
+   list(Subject1 = df[df$Subject == "1", -1],
+        Subject2 = df[df$Subject == "2", -1],
+        Subject3 = df[df$Subject == "3", -1],
+        Subject4 = df[df$Subject == "4", -1],
+        Subject5 = df[df$Subject == "5", -1],
+        Subject6 = df[df$Subject == "6", -1])
+ }
```

We will explore some ways of making this more elegant in class.

8 Getting Help in R

You’ve already learned how to get help on a particular function in *R* using `help(funcName)`. Here we’ll discuss some other aspects of *R*’s help system.

8.1 HTML Help System

In addition to “online” help accessed via `help` or `?`, *R* provides an HTML-based help system that you can access locally using your web browser. To start the help system enter:

```
> help.start()
```

There is a link “Search Engine & Keywords” on the start page of `help.start` that allows you to query the help system for a topic of interest. You can also search for help using `help.search` and `RSiteSearch`, the latter will search *R* mailing lists in addition to documentation. Finally, `apropos` is useful for finding functions that are in the current search path.

Exercise 14

Find the function for performing a Mann-Whitney test.

Solution: Executing `help.search("mann-whitney")` should lead you to `wilcox.test`.

Exercise 15

Find all the functions on your search path that have a name consisting of a single character.

Solution: See the example section of the manual page for `apropos`.

```
> apropos("^.$")
[1] "!" "$" "&" "(" "*" "+" "-" "/" ":" "<" "=" ">" "?" "@"
[15] "C" "D" "F" "H" "I" "L" "T" "W" "[" "^" "c" "i" "m" "q"
[29] "q" "t" "v" "x" "y" "z" "{" "|" "~"
```

8.2 Vignettes

Many *R* packages come with a *vignette*, a short document providing a detailed example of how to make use of the package's functionality. Vignettes contain executable *R* code that allow you to step through the examples as you read the document. You can use the `browseVignettes` function to explore the vignettes available on your system.

9 Exploring *R* objects

Every object in *R* has an associated class which you can determine using the `class` function. This is often a good way to begin exploring an unfamiliar object. Other functions useful for exploring are `length`, `dim`, `summary`, and `str`.

Exercise 16

Execute the following call and then determine what type of object is stored in `pkgs`.

```
> pkgs <- installed.packages()
```

Solution:

```
> class(pkgs)
```

```
[1] "matrix"
```

Exercise 17

What do the `length` and `dim` functions return for `pkgs`? Can you reconcile the answers given by these two functions?

Solution:

```
> ## matrices in R are stored as a vector
> ## with a dim attribute. Data is stored
> ## in column-major order.
> dp <- dim(pkgs)
> length(pkgs)
```

```
[1] 10008
```

```
> dp[1] * dp[2]
```

```
[1] 10008
```

10 Session information

- R version 2.11.1 (2010-05-31), x86_64-unknown-linux-gnu
- Locale: LC_CTYPE=C, LC_NUMERIC=C, LC_TIME=C, LC_COLLATE=C, LC_MONETARY=C, LC_MESSAGES=en_US, LC_PAPER=en_US, LC_NAME=C, LC_ADDRESS=C, LC_TELEPHONE=C, LC_MEASUREMENT=en_US, LC_IDENTIFICATION=C
- Base packages: base, datasets, grDevices, graphics, methods, stats, tools, utils
- Other packages: AnnotationDbi 1.10.2, Biobase 2.8.0, Biostrings 2.16.9, DBI 0.2-5, EatonEtAlChIPseq 0.0.1, GO.db 2.4.1, GenomicFeatures 1.0.6, GenomicRanges 1.0.7, HTSandGeneCentricLabs 0.0.3, IRanges 1.6.11, KEGG.db 2.4.1, RCurl 1.4-3, RSQLite 0.9-2, Rsamtools 1.0.7, ShortRead 1.6.2, bitops 1.0-4.1, hgu95av2.db 2.4.1, lattice 0.18-8, org.Hs.eg.db 2.4.1, rtracklayer 1.8.1
- Loaded via a namespace (and not attached): BSgenome 1.16.5, XML 3.1-0, biomaRt 2.4.0, grid 2.11.1, hwriter 1.2