Chapter 7. The Data Frame

The R equivalent of the spreadsheet.

I. Introduction

Most analytical work involves importing data from outside of R and carrying out various manipulations, tests, and visualizations. In order to complete these tasks, we need to understand how data is stored in R and how it can be accessed. Once we have a grasp of this we can consider how it can be imported (see Chapter 8).

II. Data Frames

We've already seen how R can store various kinds of data in vectors. But what happens if we have a mix of numeric and character values? One option is a *list*

```
a <- list(c(1, 2, 3), "Blue", factor(c("A", "B", "A", "B", "B")))
a
```

```
# [[1]]
# [1] 1 2 3
#
# [[2]]
# [1] "Blue"
#
# [[3]]
# [1] A B A B B
# Levels: A B
```

Notice the `[[]]` here - this is the *list element* operator. A list in R can contain an arbitrary number of items (which can be vectors) which can be of different forms - here we have one numeric, one character, an one factor, and they are all of different lengths.

A list like this may not be something you are likely to want to use often, but in most of the work you will do in R, you will be working with data that is stored as a *data frame* - this is R's most common data structure. A data frame is a special type of list - it is a list of vectors that have the same length, and whose elements correspond to one another - i.e. the 4th element of each vector correspond. Think of it like a small table in a spreadsheet, with the columns corresponding to each vector, and the rows to each record.

There are several different ways to interact with data frames in R. "Built in" data sets are stored as data frames and can be loaded with the function `data()`. External data can be read into data frames with the function `read.table()` and its relative (as we'll see in the next chapter). Existing data can be converted into a data frame using the function `data.frame()`.

```
cyl<-factor(scan(text=
    "6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4"))
am<-factor(scan(text=
    "1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1"))
levels(am)<-c("auto","manual")
disp<-scan(text=
    "2.62 2.62 1.77 4.23 5.90 3.69 5.90 2.40 2.31 2.75 2.75 4.52 4.52 4.52
    7.73 7.54 7.21 1.29 1.24 1.17 1.97 5.21 4.98 5.74 6.55 1.29 1.97 1.56
    5.75 2.38 4.93 1.98")
hp<-scan(text=
    "110 110  93 110 175 105 245 62 95 123 123 180 180 180 205 215 230 66
    52 65 97 150 150 245 175 66 91 113 264 175 335 109")
```

Here we've re-created the data on cars that we used in the last chapter.

```
car <- data.frame(cyl, disp, hp, am)
head(car)
```

```
#   cyl disp  hp     am
# 1   6 2.62 110 manual
# 2   6 2.62 110 manual
# 3   4 1.77  93 manual
# 4   6 4.23 110   auto
# 5   8 5.90 175   auto
# 6   6 3.69 105   auto
```

```
summary(car)
```

```
#  cyl         disp             hp             am
#  4:11   Min.   :1.170   Min.   : 52.0   auto  :19
#  6: 7   1st Qu.:1.978   1st Qu.: 96.5   manual:13
#  8:14   Median :3.220   Median :123.0
#         Mean   :3.781   Mean   :146.7
#         3rd Qu.:5.343   3rd Qu.:180.0
#         Max.   :7.730   Max.   :335.0
```

Now we've created a data frame named `car`. The function `head()` shows us the first 6 rows (by default). Here we see that `summary()`, when called on a data frame, gives the appropriate type of summary for each variable. The variables within the data frame have names, and we can use the function `names()` to retrieve or change these.

```
names(car)
```

```
# [1] "cyl"  "disp" "hp"   "am"
```

```
names(car)[4] <- "trans"
names(car)
```

```
# [1] "cyl"   "disp"  "hp"    "trans"
```

```
car$am
```

```
# NULL
```

```
car$trans
```

```
#  [1] manual manual manual auto   auto   auto   auto   auto   auto   auto
# [11] auto   auto   auto   auto   auto   auto   auto   manual manual manual
# [21] auto   auto   auto   auto   auto   manual manual manual manual manual
# [31] manual manual
# Levels: auto manual
```

Data in data frames can be accessed in several ways. We can use the indexing operator `[]` to access parts of a data frame by rows and columns. We can also call variables in a data frame by name using the `$` operator.

```
car[1:3, ]
```

```
#   cyl disp  hp  trans
# 1   6 2.62 110 manual
# 2   6 2.62 110 manual
# 3   4 1.77  93 manual
```

```
car[, 3]
```

```
#  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
# [18]  66  52  65  97 150 150 245 175  66  91 113 264 175 335 109
```

```
car$hp
```

```
#  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
# [18]  66  52  65  97 150 150 245 175  66  91 113 264 175 335 109
```

Note that when indexing a data frame we use 2 indices, separated by a comma (e.g. `[2,3]`). Leaving one value blank implies "all rows" or "all columns". Here the first line gives us rows 1:3, the second and third both give us the `hp` variable.

Where we've created a new data frame in this way it is important to note that *R has copied the vectors that make up the data frame*. So now we have `hp` and `car$hp`. It is important to know this because if we change one, the other is not changed.

```
hp[1] == car$hp[1]
```

```
# [1] TRUE
```

```
hp[1] <- 112
hp[1] == car$hp[1]
```

```
# [1] FALSE
```

In a case like this, it might be a good idea to *remove* the vectors we used to make the data frame, just to reduce the possibility of confusion. We can do this using the function `rm()`.

```
ls()
```

```
#  [1] "a"      "am"     "b"      "car"    "cols"   "cyl"    "die"
#  [8] "disp"   "hp"     "i"      "m.r"    "model"  "model2" "mp3"
# [15] "mtcars" "op"     "sms"    "t"      "tab"    "x"      "x.exp"
# [22] "y"      "z"
```

```
rm(cyl, disp, hp, am)
ls()
```

```
#  [1] "a"      "b"      "car"    "cols"   "die"    "i"      "m.r"
#  [8] "model"  "model2" "mp3"    "mtcars" "op"     "sms"    "t"
# [15] "tab"    "x"      "x.exp"  "y"      "z"
```

Now these vectors are no longer present in our workspace.

It is useful to know that many R functions (`lm()` for one) will accept a `data` argument - so rather than `lm(car$hp~car$cyl)` we can use `lm(hp~cyl,data=car)`. When we specify more complex models, this is very useful. Another approach is to use the function `with()` - the basic syntax is `with(some-data-frame, do-something)` - e.g. `with(car,plot(cyl,hp))`.

Indexing Data Frames Since our data `car` is a *2-dimensional* object, we ought to use 2 indices. Using the incorrect number of indices can either cause errors or cause unpleasant surprises. For example, `car[,4]` will return the 4th column, as will `car$am` or `car[[4]]`. However `car[4]` will also return the 4th column. If you had intended the 4th row (`car[4,]`) and forgotten the comma, this could cause some surprises.

```
car[[4]]
```

```
#  [1] manual manual manual auto   auto   auto   auto   auto   auto   auto
# [11] auto   auto   auto   auto   auto   auto   auto   manual manual manual
# [21] auto   auto   auto   auto   auto   manual manual manual manual manual
# [31] manual manual
# Levels: auto manual
```

```
head(car[4])
```

```
#    trans
# 1 manual
# 2 manual
# 3 manual
# 4   auto
# 5   auto
# 6   auto
```

However, if we use a single index greater than the number of columns in a data frame, R will throw an error that suggests we have selected *rows* but not columns.

```
car[5]
```

```
# Error in `[.data.frame`(car, 5): undefined columns selected
```

Similarly, if we try to call for 2 indices on a one-dimensional object (vector) we get an "incorrect number of dimensions".

```
car$hp[2, 3]
```

```
# Error in car$hp[2, 3]: incorrect number of dimensions
```

In my experience, these are rather common errors (at least for me!), and you should recognize them.

The function `subset()` is very useful for working with dataframes, since it allows you to extract data from the dataframe based on multiple conditions, and it has an easy to read syntax. For example, we can extract all the records of the `faithful` data with eruptions less than 3 minutes long (`summary()` used here to avoid spewing data over the page).

```
data(faithful)
summary(subset(faithful, eruptions <= 3))
```

```
#    eruptions        waiting
#  Min.   :1.600   Min.   :43.00
#  1st Qu.:1.833   1st Qu.:50.00
#  Median :1.983   Median :54.00
#  Mean   :2.038   Mean   :54.49
#  3rd Qu.:2.200   3rd Qu.:59.00
#  Max.   :2.900   Max.   :71.00
```

III. Attaching data

Many R tutorials will use the function `attach()` to *attach* data to the search path in R. This allows us to call variables by name. For example, in this case we have our data frame `car`, but to get the data in `hp` we need to use `car$hp` - any function that calls `hp` directly won't work - try `mean(hp)`. If we use `attach(car)` then typing `hp` gets us the data, and function calls like `mean(hp)` will now work. There are (in my experience) 2 problems with this:

A) When attaching data, R makes copies of it, so if we change `hp`, the *copy* is changed, but the original data, `car$hp` isn't changed unless we explicitly assign it to be changed - i.e. `hp[2]=NA` is *not the same* as `car$hp[2]=NA`. Read that again - `hp` is *not necessarily* the same as `car$hp`! THIS IS A VERY GOOD REASON NOT TO ATTACH DATA.

```
attach(car)
mean(hp)
```

```
# [1] 146.6875
```

```
hp[1] <- 500
hp[1] == car$hp[1]
```

```
# [1] FALSE
```

B) In my experience it is not uncommon to have multiple data sets that have many of the same variable names (e.g. `biomass`). When attaching, these conflicting names cause even more confusion. For example, if we had *not* removed the vectors `cyl, disp, and hp` above, then when we try `attach(car)` R will give us this message:

```
The following object is masked _by_ .GlobalEnv:

    cyl, disp, hp
```

For these reasons I view `attach()` as a convenience for *demonstration* of R use, and not as a "production" tool. I do not use (or only very rarely) `attach()`, and *when I do I am sure to use* ***`detach()`*** as soon as I am done with the data.

IV. Changing Data Frames

Having imported or created a data frame it is likely that we may want to *alter* it in some way. It is rather simple to remove rows or columns by indexing - `car<-car[-31,]` will *remove* the 31st row of the data and assign the data to its previous name. Similarly `car[,-4]` would remove the 4th column (though here the changed data was not assigned).

It is also very simple to add new columns (or rows) to a data frame - simply index the row (or column) `n+1`, where `n` is the number of rows (or columns). Alternately, just specifying a new name for a variable can create a new column. Here we'll demonstrate both - to calculate a new variable, displacement per cylinder, we first need cylinders as numeric. We'll use the 'approved' method of converting a factor to numeric - indexing the levels (see Chapter 2).

```
car[, 5] <- as.numeric(levels(car$cyl)[car$cyl])
names(car)
```

```
# [1] "cyl"   "disp"  "hp"    "trans" "V5"
```

```
names(car)[5] <- "cyl.numeric"
```

Our data set now has 5 columns, but until we give the new variable a name it is just `"V5"`, for 'Variable 5'. Let's calculate displacement per cylinder:

```
car$disp.per.cyl <- car$disp/car$cyl.numeric
names(car)
```

```
# [1] "cyl"          "disp"         "hp"           "trans"
# [5] "cyl.numeric"  "disp.per.cyl"
```

This method of creating a new variable is easier because we don't have to bother about the variable name, or about which column in will occupy. Had we used a numeric index of 5, we would overwrite the value in that column.

Sometimes we might wish to combine 2 data frames together. We can do this using `cbind()` and `rbind()` (for *column* -wise and *row* -wise binding respectively). The dataset `mtcars` contains several variables that are not in our data frame `cars`. We'll use `cbind()` to combine the 2 data sets.

```
data(mtcars)  # load the data
names(mtcars)  # cols 1,5:8,10:11 not in our data
```

```
#  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
# [11] "carb"
```

```
dim(car)
```

```
# [1] 32  6
```

```
car <- cbind(car, mtcars[, c(1, 5:8, 10:11)])
dim(car)
```

```
# [1] 32 13
```

```
head(car)
```

```
#                   cyl disp  hp  trans cyl.numeric disp.per.cyl  mpg drat
# Mazda RX4           6 2.62 110 manual           6    0.4366667 21.0 3.90
# Mazda RX4 Wag       6 2.62 110 manual           6    0.4366667 21.0 3.90
# Datsun 710          4 1.77  93 manual           4    0.4425000 22.8 3.85
# Hornet 4 Drive      6 4.23 110   auto           6    0.7050000 21.4 3.08
# Hornet Sportabout   8 5.90 175   auto           8    0.7375000 18.7 3.15
# Valiant             6 3.69 105   auto           6    0.6150000 18.1 2.76
#                     wt  qsec vs gear carb
# Mazda RX4         2.620 16.46  0    4    4
# Mazda RX4 Wag     2.875 17.02  0    4    4
# Datsun 710        2.320 18.61  1    4    1
# Hornet 4 Drive    3.215 19.44  1    3    1
# Hornet Sportabout 3.440 17.02  0    3    2
# Valiant           3.460 20.22  1    3    1
```

Note that the row names from `mtcars` have followed the variables from that data frame. A couple of observations about using adding to data frames: * *Don't use* ***`cbind()`*** *if the rows don't correspond!*- we'll see how to use `merge()` (Chapter 10) which is the right tool for this situation. (Similarly don't use `rbind()` if the columns don't correspond). * `cbind()` and `rbind()` are rather slow - don't use them inside a loop! * If you are writing a loop it is far more efficient to make space for your output (whether in a new data frame or by adding to one) *before* the loop begins, adding a row to your data frame in each iteration of a loop will slow your code down.

EXTRA: Comments There is an attribute of data frames that is reserved for comments. The function `comment()` allows one to set this. `comment(car)` will return `NULL` because no comment has been set, but we can use the same function to set comments.

```
comment(car) <- "A data set derived from the mtcars dataset. Displacement is in liters"
```

Now we have added a comment to this dataset, and `comment(car)` will retrieve it.

V. Exercises

1) Use the `mtcars` data (`data(mtcars)`) to answer these questions (if you get confused, review the bit on logical extraction in Chapter 1):
a) Which rows of the data frame contain cars that weigh more than 4000 pounds (the variable is `wt`, units are 1000 pounds).
b) Which cars are these? (*Hint:* since rows are named by car name, use `row.names()`).
c) What is the mean displacement (in inches^3) for cars with at least 200 horsepower (`hp`).
d) Which car has the highest fuel economy (`mpg`)?
e) What was the fuel economy for the Honda Civic?

2) Using the `mtcars` data create a new variable for horsepower per unit weight (`hp/wt`). Is this a better predictor of acceleration (`qsec`; seconds to complete a quarter mile) than raw horsepower? (Hint - check out correlations between these variables and acceleration, or fit regressions for both models).

3) Use the function `subset()` to return the cars with 4 cylinders and automatic transmissions (`am = 0`). (*Hint:* use "&" for logical "AND"; see `?Logic` and select `Logical Operators`).