Chapter 18: Data frames

Section 18.1: Create an empty data.frame

A data.frame is a special kind of list: it is *rectangular*. Each element (column) of the list has same length, and where each row has a "row name". Each column has its own class, but the class of one column can be different from the class of another column (unlike a matrix, where all elements must have the same class).

In principle, a data.frame could have no rows and no columns:

```
> structure(list(character()), class = "data.frame")
NULL
<0 rows> (or 0-length row.names)
```

But this is unusual. It is more common for a data.frame to have many columns and many rows. Here is a data.frame with three rows and two columns (`a` is numeric class and `b` is character class):

```
> structure(list(a = 1:3, b = letters[1:3]), class = "data.frame")
[1] a b
<0 rows> (or 0-length row.names)
```

In order for the data.frame to print, we will need to supply some row names. Here we use just the numbers 1:3:

```
> structure(list(a = 1:3, b = letters[1:3]), class = "data.frame", row.names = 1:3)
  a b
1 1 a
2 2 b
3 3 c
```

Now it becomes obvious that we have a data.frame with 3 rows and 2 columns. You can check this using `nrow()`, `ncol()`, and `dim()`:

```
> x <- structure(list(a = numeric(3), b = character(3)), class = "data.frame", row.names = 1:3)
> nrow(x)
[1] 3
> ncol(x)
[1] 2
> dim(x)
[1] 3 2
```

R provides two other functions (besides `structure()`) that can be used to create a data.frame. The first is called, intuitively, `data.frame()`. It checks to make sure that the column names you supplied are valid, that the list elements are all the same length, and supplies some automatically generated row names. This means that the output of `data.frame()` might now always be exactly what you expect:

```
> str(data.frame("a a a" = numeric(3), "b-b-b" = character(3)))
'data.frame':   3 obs. of  2 variables:
 $ a.a.a: num  0 0 0
 $ b.b.b: Factor w/ 1 level "": 1 1 1
```

The other function is called `as.data.frame()`. This can be used to coerce an object that is not a data.frame into being a data.frame by running it through `data.frame()`. As an example, consider a matrix:

```
> m <- matrix(letters[1:9], nrow = 3)
> m
```

```
     [,1] [,2] [,3]
[1,] "a"  "d"  "g"
[2,] "b"  "e"  "h"
[3,] "c"  "f"  "i"
```

And the result:

```
> as.data.frame(m)
  V1 V2 V3
1  a  d  g
2  b  e  h
3  c  f  i
> str(as.data.frame(m))
'data.frame':   3 obs. of  3 variables:
 $ V1: Factor w/ 3 levels "a","b","c": 1 2 3
 $ V2: Factor w/ 3 levels "d","e","f": 1 2 3
 $ V3: Factor w/ 3 levels "g","h","i": 1 2 3
```

Section 18.2: Subsetting rows and columns from a data frame

Syntax for accessing rows and columns: [, [[, and $

This topic covers the most common syntax to access specific rows and columns of a data frame. These are

- Like a `matrix` with single brackets `data[rows, columns]`
  - Using row and column numbers
  - Using column (and row) names
- Like a `list`:
  - With single brackets `data[columns]` to get a data frame
  - With double brackets `data[[one_column]]` to get a vector
- With $ for a single column `data$column_name`

We will use the built-in `mtcars` data frame to illustrate.

Like a matrix: `data[rows, columns]`
With numeric indexes

Using the built in data frame `mtcars`, we can extract rows and columns using [ ] brackets with a comma included. Indices before the comma are rows:

```
# get the first row
mtcars[1, ]
# get the first five rows
mtcars[1:5, ]
```

Similarly, after the comma are columns:

```
# get the first column
mtcars[, 1]
# get the first, third and fifth columns:
mtcars[, c(1, 3, 5)]
```

As shown above, if either rows or columns are left blank, all will be selected. `mtcars[1, ]` indicates the first row with *all* the columns.

With column (and row) names

So far, this is identical to how rows and columns of matrices are accessed. With `data.frame`s, most of the time it is preferable to use a column name to a column index. This is done by using a `character` with the column name instead of `numeric` with a column number:

```
# get the mpg column
mtcars[, "mpg"]
# get the mpg, cyl, and disp columns
mtcars[, c("mpg", "cyl", "disp")]
```

Though less common, row names can also be used:

```
mtcars["Mazda Rx4", ]
```

Rows and columns together

The row and column arguments can be used together:

```
# first four rows of the mpg column
mtcars[1:4, "mpg"]

# 2nd and 5th row of the mpg, cyl, and disp columns
mtcars[c(2, 5), c("mpg", "cyl", "disp")]
```

A warning about dimensions:

When using these methods, if you extract multiple columns, you will get a data frame back. However, if you extract a *single* column, you will get a vector, not a data frame under the default options.

```
## multiple columns returns a data frame
class(mtcars[, c("mpg", "cyl")])
# [1] "data.frame"
## single column returns a vector
class(mtcars[, "mpg"])
# [1] "numeric"
```

There are two ways around this. One is to treat the data frame as a list (see below), the other is to add a `drop = FALSE` argument. This tells R to not "drop the unused dimensions":

```
class(mtcars[, "mpg", drop = FALSE])
# [1] "data.frame"
```

Note that matrices work the same way - by default a single column or row will be a vector, but if you specify `drop = FALSE` you can keep it as a one-column or one-row matrix.

Like a list

Data frames are essentially `list`s, i.e., they are a list of column vectors (that all must have the same length). Lists can be subset using single brackets `[` for a sub-list, or double brackets `[[` for a single element.

With single brackets `data[columns]`

When you use single brackets and no commas, you will get column back because data frames are lists of columns.

```
mtcars["mpg"]
mtcars[c("mpg", "cyl", "disp")]
my_columns <- c("mpg", "cyl", "hp")
mtcars[my_columns]
```

Single brackets *like a list* vs. single brackets *like a matrix*

The difference between `data[columns]` and `data[, columns]` is that when treating the `data.frame` as a `list` (no comma in the brackets) the object returned will *be a* `data.frame`. If you use a comma to treat the `data.frame` like a `matrix` then selecting a single column will return a vector but selecting multiple columns will return a `data.frame`.

```
## When selecting a single column
## like a list will return a data frame
class(mtcars["mpg"])
# [1] "data.frame"
## like a matrix will return a vector
class(mtcars[, "mpg"])
# [1] "numeric"
```

With double brackets `data[[one_column]]`

To extract a single column *as a vector* when treating your `data.frame` as a `list`, you can use double brackets `[[`. This will only work for a single column at a time.

```
# extract a single column by name as a vector
mtcars[["mpg"]]

# extract a single column by name as a data frame (as above)
mtcars["mpg"]
```

Using $ to access columns

A single column can be extracted using the magical shortcut $ without using a quoted column name:

```
# get the column "mpg"
mtcars$mpg
```

Columns accessed by $ will always be vectors, not data frames.

Drawbacks of $ for accessing columns

The $ can be a convenient shortcut, especially if you are working in an environment (such as RStudio) that will auto-complete the column name in this case. **However,** $ has drawbacks as well: it uses *non-standard evaluation* to avoid the need for quotes, which means it *will not work* if your column name is stored in a variable.

```
my_column <- "mpg"
# the below will not work
mtcars$my_column
# but these will work
mtcars[, my_column]  # vector
mtcars[my_column]    # one-column data frame
mtcars[[my_column]]  # vector
```

Due to these concerns, $ is best used in *interactive* R sessions when your column names are constant. For *programmatic* use, for example in writing a generalizable function that will be used on different data sets with different column names, $ should be avoided.

Also note that the default behaviour is to use partial matching only when extracting from recursive objects (except environments) by $

```
# give you the values of "mpg" column
# as "mtcars" has only one column having name starting with "m"
```

```r
mtcars$m
# will give you "NULL"
# as "mtcars" has more than one columns having name starting with "d"
mtcars$d
```

Advanced indexing: negative and logical indices

Whenever we have the option to use numbers for a index, we can also use negative numbers to omit certain indices or a boolean (logical) vector to indicate exactly which items to keep.

Negative indices omit elements

```r
mtcars[1, ]   # first row
mtcars[ -1, ] # everything but the first row
mtcars[-(1:10), ] # everything except the first 10 rows
```

Logical vectors indicate specific elements to keep

We can use a condition such as `<` to generate a logical vector, and extract only the rows that meet the condition:

```r
# logical vector indicating TRUE when a row has mpg less than 15
# FALSE when a row has mpg >= 15
test <- mtcars$mpg < 15

# extract these rows from the data frame
mtcars[test, ]
```

We can also bypass the step of saving the intermediate variable

```r
# extract all columns for rows where the value of cyl is 4.
mtcars[mtcars$cyl == 4, ]
# extract the cyl, mpg, and hp columns where the value of cyl is 4
mtcars[mtcars$cyl == 4, c("cyl", "mpg", "hp")]
```

Section 18.3: Convenience functions to manipulate data.frames

Some convenience functions to manipulate `data.frames` are **`subset()`**, **`transform()`**, **`with()`** and **`within()`**.

subset

The **`subset()`** function allows you to subset a **`data.frame`** in a more convenient way (subset also works with other classes):

```r
subset(mtcars, subset = cyl == 6, select = c("mpg", "hp"))
                mpg  hp
Mazda RX4      21.0 110
Mazda RX4 Wag  21.0 110
Hornet 4 Drive 21.4 110
Valiant        18.1 105
Merc 280       19.2 123
Merc 280C      17.8 123
Ferrari Dino   19.7 175
```

In the code above we asking only for the lines in which `cyl == 6` and for the columns `mpg` and `hp`. You could achieve the same result using [ ] with the following code:

```r
mtcars[mtcars$cyl == 6, c("mpg", "hp")]
```

transform

The `transform()` function is a convenience function to change columns inside a `data.frame`. For instance the following code adds another column named `mpg2` with the result of `mpg^2` to the `mtcars` `data.frame`:

```
mtcars <- transform(mtcars, mpg2 = mpg^2)
```

with and within

Both `with()` and `within()` let you to evaluate expressions inside the `data.frame` environment, allowing a somewhat cleaner syntax, saving you the use of some `$` or `[]`.

For example, if you want to create, change and/or remove multiple columns in the `airquality` `data.frame`:

```
aq <- within(airquality, {
    lOzone <- log(Ozone) # creates new column
    Month <- factor(month.abb[Month]) # changes Month Column
    cTemp <- round((Temp - 32) * 5/9, 1) # creates new column
    S.cT <- Solar.R / cTemp  # creates new column
    rm(Day, Temp) # removes columns
})
```

Section 18.4: Introduction

Data frames are likely the data structure you will used most in your analyses. A data frame is a special kind of list that stores same-length vectors of different classes. You create data frames using the `data.frame` function. The example below shows this by combining a numeric and a character vector into a data frame. It uses the `:` operator, which will create a vector containing all integers from 1 to 3.

```
df1 <- data.frame(x = 1:3, y = c("a", "b", "c"))
df1
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
class(df1)
## [1] "data.frame"
```

Data frame objects do not print with quotation marks, so the class of the columns is not always obvious.

```
df2 <- data.frame(x = c("1", "2", "3"), y = c("a", "b", "c"))
df2
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
```

Without further investigation, the "x" columns in `df1` and `df2` cannot be differentiated. The `str` function can be used to describe objects with more detail than class.

```
str(df1)
## 'data.frame':    3 obs. of  2 variables:
##  $ x: int  1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
str(df2)
## 'data.frame':    3 obs. of  2 variables:
```

```
##  $ x: Factor w/ 3 levels "1","2","3": 1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Here you see that `df1` is a **`data.frame`** and has 3 observations of 2 variables, "x" and "y." Then you are told that "x" has the data type integer (not important for this class, but for our purposes it behaves like a numeric) and "y" is a factor with three levels (another data class we are not discussing). ***It is important to note that, by default, data frames coerce characters to factors.*** The default behavior can be changed with the `stringsAsFactors` parameter:

```
df3 <- data.frame(x = 1:3, y = c("a", "b", "c"), stringsAsFactors = FALSE)
str(df3)
## 'data.frame':    3 obs. of  2 variables:
##  $ x: int  1 2 3
##  $ y: chr  "a" "b" "c"
```

Now the "y" column is a character. As mentioned above, each "column" of a data frame must have the same length. Trying to create a data.frame from vectors with different lengths will result in an error. (Try running `data.frame(x = 1:3, y = 1:4)` to see the resulting error.)

As test-cases for data frames, some data is provided by R by default. One of them is iris, loaded as follows:

```
mydataframe <- iris
str(mydataframe)
```

Section 18.5: Convert all columns of a data.frame to character class

A common task is to convert all columns of a data.frame to character class for ease of manipulation, such as in the cases of sending data.frames to a RDBMS or merging data.frames containing factors where levels may differ between input data.frames.

The best time to do this is when the data is read in - almost all input methods that create data frames have an options `stringsAsFactors` which can be set to `FALSE`.

If the data has already been created, factor columns can be converted to character columns as shown below.

```
bob <- data.frame(jobs = c("scientist", "analyst"),
                  pay  = c(160000, 100000), age = c(30, 25))
str(bob)
```

```
'data.frame':    2 obs. of  3 variables:
 $ jobs: Factor w/ 2 levels "analyst","scientist": 2 1
 $ pay : num  160000 100000
 $ age : num  30 25
```

```
# Convert *all columns* to character
bob[] <- lapply(bob, as.character)
str(bob)
```

```
    'data.frame':    2 obs. of  3 variables:
 $ jobs: chr  "scientist" "analyst"
 $ pay : chr  "160000" "1e+05"
 $ age : chr  "30" "25"
```

```
# Convert only factor columns to character
bob[] <- lapply(bob, function(x) {
```

```
    if is.factor(x) x <- as.character(x)
    return(x)
})
```