17.5: Extracting a Subset of a Data Frame

In this section we turn to the question of how to subset a data frame rather than a vector. To that end, the first thing I should point out is that, if all you want to do is subset *one* of the variables inside the data frame, then as usual the `$` operator is your friend. For instance, suppose I'm working with the `itng` data frame, and what I want to do is create the `speech.by.char` list. I can use the exact same tricks that I used last time, since what I really want to do is `split()` the `itng$utterance` vector, using the `itng$speaker` vector as the grouping variable. However, most of the time what you actually want to do is select several different variables within the data frame (i.e., keep only some of the columns), or maybe a subset of cases (i.e., keep only some of the rows). In order to understand how this works, we need to talk more specifically about data frames and how to subset them.

Using the `subset()` function

There are several different ways to subset a data frame in R, some easier than others. I'll start by discussing the `subset()` function, which is probably the conceptually simplest way do it. For our purposes there are three different arguments that you'll be most interested in:

- `x` . The data frame that you want to subset.
- `subset` . A vector of logical values indicating which cases (rows) of the data frame you want to keep. By default, all cases will be retained.
- `select` . This argument indicates which variables (columns) in the data frame you want to keep. This can either be a list of variable names, or a logical vector indicating which ones to keep, or even just a numeric vector containing the relevant column numbers. By default, all variables will be retained.

Let's start with an example in which I use all three of these arguments. Suppose that I want to subset the `itng` data frame, keeping only the utterances made by Makka-Pakka. What that means is that I need to use the `select` argument to pick out the `utterance` variable, and I also need to use the `subset` variable, to pick out the cases when Makka-Pakka is speaking (i.e., `speaker == "makka-pakka"`). Therefore, the command I need to use is this:

```
df <- subset( x = itng,                          # data frame is itng
              subset = speaker == "makka-pakka",  # keep only Makka-Pakkas speech
              select = utterance )                # keep only the utterance variable
print( df )
```

```
##    utterance
## 7        pip
## 8        pip
## 9        onk
## 10       onk
```

The variable `df` here is still a data frame, but it only contains one variable (called `utterance`) and four cases. Notice that the row numbers are actually the same ones from the original data frame. It's worth taking a moment to briefly explain this. The reason that this happens is that these "row numbers' are actually row *names*. When you create a new data frame from scratch R will assign each row a fairly boring row name, which is identical to the row number. However, when you subset the data frame, each row keeps its original row name. This can be quite useful, since – as in the current example – it provides you a visual reminder of what each row in the new data frame corresponds to in the original data frame. However, if it annoys you, you can change the row names using the `rownames()` function.[113]

In any case, let's return to the `subset()` function, and look at what happens when we don't use all three of the arguments. Firstly, suppose that I didn't bother to specify the `select` argument. Let's see what happens:

```
subset( x = itng,
        subset = speaker == "makka-pakka" )
```

```
##        speaker utterance
## 7  makka-pakka       pip
## 8  makka-pakka       pip
## 9  makka-pakka       onk
## 10 makka-pakka       onk
```

Not surprisingly, R has kept the same cases from the original data set (i.e., rows 7 through 10), but this time it has kept all of the variables from the data frame. Equally unsurprisingly, if I don't specify the `subset` argument, what we find is that R keeps all of the cases:

```
subset( x = itng,
        select = utterance )
```

```
##    utterance
## 1        pip
## 2        pip
## 3        onk
## 4        onk
## 5         ee
## 6         oo
## 7        pip
## 8        pip
## 9        onk
## 10       onk
```

Again, it's important to note that this output is still a data frame: it's just a data frame with only a single variable.

Using square brackets: I. Rows and columns

Throughout the book so far, whenever I've been subsetting a vector I've tended use the square brackets `[]` to do so. But in the previous section when I started talking about subsetting a data frame I used the `subset()` function. As a consequence, you might be wondering whether it is possible to use the square brackets to subset a data frame. The answer, of course, is yes. Not only *can* you use square brackets for this purpose, as you become more familiar with R you'll find that this is actually much more convenient than using `subset()` . Unfortunately, the use of square brackets for this purpose is somewhat complicated, and can be very confusing to novices. So be warned: this section is more complicated than it feels like it "should" be. With that warning in place, I'll try to walk you through it slowly. For this section, I'll use a slightly different data set, namely the `garden` data frame that is stored in the `"nightgarden2.Rdata"` file.

```
load("./rbook-master/data/nightgarden2.Rdata" )
garden
```

```
##           speaker utterance line
## case.1  upsy-daisy       pip    1
## case.2  upsy-daisy       pip    2
## case.3   tombliboo        ee    5
## case.4 makka-pakka       pip    7
## case.5 makka-pakka       onk    9
```

As you can see, the `garden` data frame contains 3 variables and 5 cases, and this time around I've used the `rownames()` function to attach slightly verbose labels to each of the cases. Moreover, let's assume that what we want to do is to pick out rows 4 and 5 (the two cases when Makka-Pakka is speaking), and columns 1 and 2 (variables `speaker` and `utterance`).

How shall we do this? As usual, there's more than one way. The first way is based on the observation that, since a data frame is basically a table, every element in the data frame has a row number and a column number. So, if we want to pick out a single element, we have to specify the row number *and* a column number within the square brackets. By convention, the row number comes first. So, for the data frame above, which has 5 rows and 3 columns, the numerical indexing scheme looks like this:

```
knitr::kable(data.frame(stringsAsFactors=FALSE, row = c("1","2","3", "4", "5"), col1
"[5,1]"), col2 = c("[1,2]", "[2,2]", "[3,2]", "[4,2]", "[5,2]"), col3 = c("[1,3]", "
```

row	col1	col2	col3
1	[1,1]	[1,2]	[1,3]
2	[2,1]	[2,2]	[2,3]
3	[3,1]	[3,2]	[3,3]
4	[4,1]	[4,2]	[4,3]
5	[5,1]	[5,2]	[5,3]

If I want the 3rd case of the 2nd variable, what I would type is `garden[3,2]`, and R would print out some output showing that, this element corresponds to the utterance `"ee"`. However, let's hold off from actually doing that for a moment, because there's something slightly counterintuitive about the specifics of what R does under those circumstances (see Section 7.5.4). Instead, let's aim to solve our original problem, which is to pull out two rows (4 and 5) and two columns (1 and 2). This is fairly simple to do, since R allows us to specify multiple rows and multiple columns. So let's try that:

```
garden[ 4:5, 1:2 ]
```

```
##           speaker utterance
## case.4 makka-pakka       pip
## case.5 makka-pakka       onk
```

Clearly, that's exactly what we asked for: the output here is a data frame containing two variables and two cases. Note that I could have gotten the same answer if I'd used the `c()` function to produce my vectors rather than the `:` operator. That is, the following command is equivalent to the last one:

```
garden[ c(4,5), c(1,2) ]
```

```
##           speaker utterance
## case.4 makka-pakka       pip
## case.5 makka-pakka       onk
```

It's just not as pretty. However, if the columns and rows that you want to keep don't happen to be next to each other in the original data frame, then you might find that you have to resort to using commands like `garden[ c(2,4,5), c(1,3) ]` to extract them.

A second way to do the same thing is to use the names of the rows and columns. That is, instead of using the row numbers and column numbers, you use the character strings that are used as the labels for the rows and columns. To apply this idea to our `garden` data frame, we would use a command like this:

```
garden[ c("case.4", "case.5"), c("speaker", "utterance") ]
```

```
##            speaker utterance
## case.4 makka-pakka       pip
## case.5 makka-pakka       onk
```

Once again, this produces exactly the same output, so I haven't bothered to show it. Note that, although this version is more annoying to *type* than the previous version, it's a bit easier to *read*, because it's often more meaningful to refer to the elements by their names rather than their numbers. Also note that you don't have to use the same convention for the rows and columns. For instance, I often find that the variable names are meaningful and so I sometimes refer to them by name, whereas the row names are pretty arbitrary so it's easier to refer to them by number. In fact, that's more or less exactly what's happening with the `garden` data frame, so it probably makes more sense to use this as the command:

```
garden[ 4:5, c("speaker", "utterance") ]
```

```
##            speaker utterance
## case.4 makka-pakka       pip
## case.5 makka-pakka       onk
```

Again, the output is identical.

Finally, both the rows and columns can be indexed using logicals vectors as well. For example, although I *claimed* earlier that my goal was to extract cases 4 and 5, it's pretty obvious that what I really wanted to do was select the cases where Makka-Pakka is speaking. So what I could have done is create a logical vector that indicates which cases correspond to Makka-Pakka speaking:

```
is.MP.speaking <- garden$speaker == "makka-pakka"
is.MP.speaking
```

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

As you can see, the 4th and 5th elements of this vector are `TRUE` while the others are `FALSE`. Now that I've constructed this "indicator" variable, what I can do is use this vector to select the rows that I want to keep:

```
garden[ is.MP.speaking, c("speaker", "utterance") ]
```

```
##            speaker utterance
## case.4 makka-pakka       pip
## case.5 makka-pakka       onk
```

And of course the output is, yet again, the same.

Using square brackets: II. Some elaborations

There are two fairly useful elaborations on this "rows and columns" approach that I should point out. Firstly, what if you want to keep all of the rows, or all of the columns? To do this, all we have to do is leave the corresponding entry blank, but it is crucial to remember to keep the comma*! For instance, suppose I want to keep all the rows in the `garden` data, but I only want to retain the first two columns. The easiest way do this is to use a command like this:

```
garden[ , 1:2 ]
```

```
##            speaker utterance
## case.1  upsy-daisy       pip
## case.2  upsy-daisy       pip
## case.3   tombliboo        ee
## case.4 makka-pakka       pip
## case.5 makka-pakka       onk
```

Alternatively, if I want to keep all the columns but only want the last two rows, I use the same trick, but this time I leave the second index blank. So my command becomes:

```
garden[ 4:5, ]
```

```
##            speaker utterance line
## case.4 makka-pakka       pip    7
## case.5 makka-pakka       onk    9
```

The second elaboration I should note is that it's still okay to use negative indexes as a way of telling R to delete certain rows or columns. For instance, if I want to delete the 3rd column, then I use this command:

```
garden[ , -3 ]
```

```
##            speaker utterance
## case.1  upsy-daisy       pip
## case.2  upsy-daisy       pip
## case.3   tombliboo        ee
## case.4 makka-pakka       pip
## case.5 makka-pakka       onk
```

whereas if I want to delete the 3rd row, then I'd use this one:

```
garden[ -3,  ]
```

```
##            speaker utterance line
## case.1  upsy-daisy       pip    1
## case.2  upsy-daisy       pip    2
## case.4 makka-pakka       pip    7
## case.5 makka-pakka       onk    9
```

So that's nice.

Using square brackets: III. Understanding "dropping"

At this point some of you might be wondering why I've been so terribly careful to choose my examples in such a way as to ensure that the output always has are multiple rows and multiple columns. The reason for this is that I've been trying to hide the somewhat curious "dropping" behaviour that R produces when the output only has a single column. I'll start by showing you what happens, and then I'll try to explain it. Firstly, let's have a look at what happens when the output contains only a single *row*:

```
garden[ 5, ]
```

```
##            speaker utterance line
## case.5 makka-pakka       onk    9
```

This is exactly what you'd expect to see: a data frame containing three variables, and only one case per variable. Okay, no problems so far. What happens when you ask for a single *column*? Suppose, for instance, I try this as a command:

```
garden[ , 3 ]
```

Based on everything that I've shown you so far, you would be well within your rights to expect to see R produce a data frame containing a single variable (i.e., `line`) and five cases. After all, that *is* what the `subset()` command does in this situation, and it's pretty consistent with everything else that I've shown you so far about how square brackets work. In other words, you should expect to see this:

```
         line
case.1    1
case.2    2
case.3    5
case.4    7
case.5    9
```

However, that is emphatically not what happens. What you actually get is this:

```
garden[ , 3 ]
```

```
## [1] 1 2 5 7 9
```

That output is *not a data frame* at all! That's just an ordinary numeric vector containing 5 elements. What's going on here is that R has "noticed" that the output that we've asked for doesn't really "need" to be wrapped up in a data frame at all, because it only corresponds to a single variable. So what it does is "drop" the output from a data frame *containing* a single variable, "down" to a simpler output that corresponds to that variable. This behaviour is actually very convenient for day to day usage once you've become familiar with it – and I suppose that's the real reason why R does this – but there's no escaping the fact that it is *deeply* confusing to novices. It's especially confusing because the behaviour appears only for a very specific case: (a) it only works for columns and not for rows, because the columns correspond to variables and the rows do not, and (b) it only applies to the "rows and columns" version of the square brackets, and not to the `subset()` function,[114] or to the "just columns" use of the square brackets (next section). As I say, it's very confusing when you're just starting out. For what it's worth, you can suppress this behaviour if you want, by setting `drop = FALSE` when you construct your bracketed expression. That is, you could do something like this:

```
garden[ , 3, drop = FALSE ]
```

```
##         line
## case.1    1
## case.2    2
## case.3    5
## case.4    7
## case.5    9
```

I suppose that helps a little bit, in that it gives you some control over the dropping behaviour, but I'm not sure it helps to make things any easier to understand. Anyway, that's the "dropping" special case. Fun, isn't it?

Using square brackets: IV. Columns only

As if the weird "dropping" behaviour wasn't annoying enough, R actually provides a completely different way of using square brackets to index a data frame. Specifically, if you *only* give a single index, R will assume you want the corresponding columns, not the rows. Do not be fooled by the fact that this second method also uses square brackets: it behaves differently to the "rows and columns" method that I've discussed in the last few sections. Again, what I'll do is show you *what* happens first, and then I'll try to explain *why* it happens afterwards. To that end, let's start with the following command:

```
garden[ 1:2 ]
```

```
##            speaker utterance
## case.1  upsy-daisy       pip
## case.2  upsy-daisy       pip
## case.3   tombliboo        ee
## case.4 makka-pakka       pip
## case.5 makka-pakka       onk
```

As you can see, the output gives me the first two columns, much as if I'd typed `garden[,1:2]` . It doesn't give me the first two rows, which is what I'd have gotten if I'd used a command like `garden[1:2,]` . Not only that, if I ask for a *single* column, R does not drop the output:

```
garden[3]
```

```
##        line
## case.1    1
## case.2    2
## case.3    5
## case.4    7
## case.5    9
```

As I said earlier, the *only* case where dropping occurs by default is when you use the "row and columns" version of the square brackets, and the output happens to correspond to a single column. However, if you really want to force R to drop the output, you can do so using the "double brackets" notation:

```
garden[[3]]
```

```
## [1] 1 2 5 7 9
```

Note that R will only allow you to ask for one column at a time using the double brackets. If you try to ask for multiple columns in this way, you get completely different behaviour,[115] which may or may not produce an error, but definitely won't give you the output you're expecting. The only reason I'm mentioning it at all is that you might run into double brackets when doing further reading, and a lot of books don't explicitly point out the difference between `[` and `[[` . However, I promise that I won't be using `[[` anywhere else in this book.

Okay, for those few readers that have persevered with this section long enough to get here without having set fire to the book, I should explain *why* R has these two different systems for subsetting a data frame (i.e., "row and column" versus "just columns"), and why they behave so differently to each other. I'm not 100% sure about this since I'm still reading through some of the old references that describe the early development of R, but I think the answer relates to the fact that data frames are actually a very strange hybrid of two different kinds of thing. At a low level, a data frame is a list (Section 4.9). I can demonstrate this to you by overriding the normal `print()` function[116] and forcing R to print out the `garden` data frame using the default print method rather than the special one that is defined only for data frames. Here's what we get:

```
print.default( garden )
```

```
## $speaker
## [1] upsy-daisy  upsy-daisy  tombliboo   makka-pakka makka-pakka
## Levels: makka-pakka tombliboo upsy-daisy
##
## $utterance
## [1] pip pip ee  pip onk
## Levels: ee onk oo pip
##
## $line
## [1] 1 2 5 7 9
##
## attr(,"class")
## [1] "data.frame"
```

Apart from the weird part of the output right at the bottom, this is *identical* to the print out that you get when you print out a list (see Section 4.9). In other words, a data frame is a list. View from this "list based" perspective, it's clear what `garden[1]` is: it's the first variable stored in the list, namely `speaker` . In other words, when you use the "just columns" way of indexing a data frame, using only a single index, R assumes that you're thinking about the data frame as if it were a *list of variables*. In fact, when you use the `$` operator you're taking advantage of the fact that the data frame is secretly a list.

However, a data frame is more than just a list. It's a very special kind of list where all the variables are of the same length, and the first element in each variable happens to correspond to the first "case" in the data set. That's why no-one ever wants to see a data frame printed out in the default "list-like" way that I've shown in the extract above. In terms of the deeper *meaning* behind what a data frame is used for, a data frame really does have this rectangular shape to it:

```
print( garden )
```

```
##           speaker utterance line
## case.1  upsy-daisy       pip    1
## case.2  upsy-daisy       pip    2
## case.3   tombliboo        ee    5
## case.4 makka-pakka       pip    7
## case.5 makka-pakka       onk    9
```

Because of the fact that a data frame is basically a table of data, R provides a second "row and column" method for interacting with the data frame (see Section 7.11.1 for a related example). This method makes much more sense in terms of the high-level *table of data* interpretation of what a data frame is, and so for the most part it's this method that people tend to prefer. In fact, throughout the rest of the book I will be sticking to the "row and column" approach (though I will use `$` a lot), and never again referring to the "just columns" approach. However, it does get used a lot in practice, so I think it's important that this book explain what's going on.

And now let us never speak of this again.

This page titled 17.5: Extracting a Subset of a Data Frame is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Danielle Navarro via source content that was edited to the style and standards of the LibreTexts platform.

- **7.5: Extracting a Subset of a Data Frame** by Danielle Navarro is licensed CC BY-SA 4.0. Original source: https://bookdown.org/ekothe/navarro26/.