Everything I know about R subsetting

David Hugh-Jones

02/08/2018

R subsetting is complex but very powerful. Or, it’s powerful but very complex – so complex that some of what I am about to tell you, I learned only this year. (I’ve been using R since 2003.) This page has everything I know about it, from the basics up. If you’re an expert, there still might be something for you here: you might want to skip to the “Advanced” section.

The basics

Let’s give ourselves some variables to play with – a simple character vector with some names.

```
food <- c(
        ham      = "Ham",
        eggs     = "Eggs",
        tomatoes = "Tomatoes"
      )
```

Numeric, character and logical subsets

R knows three basic way to subset.

The first is the easiest: subsetting with a number `n` gives you the nth element.

```
food[1]
```

```
  ham
"Ham"
```

If you have a vector of numbers, you get a vector of elements.

```
food[2:3]
```

```
      eggs   tomatoes
    "Eggs" "Tomatoes"
```

```
food[c(1, 3)]
```

```
       ham   tomatoes
     "Ham" "Tomatoes"
```

The second is also pretty easy: if you subset with a character vector, you get the element(s) with the corresponding name(s).

```
food["ham"]
```

```
  ham
"Ham"
```

```
food[c("ham", "eggs")]
```

```
   ham   eggs
 "Ham" "Eggs"
```

You can repeat yourself:

```
cat(food[c("ham", "ham", "ham")], "! I love ham!")
```

```
Ham Ham Ham ! I love ham!
```

Lastly, you can subset with a logical vector. This is a bit different: it returns every element where the logical vector is `TRUE`.

```
allergies <- c(FALSE, TRUE, FALSE)
vegetarian <- c(FALSE, TRUE, TRUE)
vegan <- c(FALSE, FALSE, TRUE)

# Vegetarian food:
food[vegetarian]
```

```
      eggs   tomatoes
    "Eggs" "Tomatoes"
```

```
# Vegan food:
food[vegan]
```

```
  tomatoes
"Tomatoes"
```

Logical subsetting is useful when you want to find things with a particular characteristic:

```
# Long food names:
nchar(food) > 5
```

```
     ham     eggs tomatoes
   FALSE    FALSE     TRUE
```

```
food[nchar(food) > 5]
```

```
  tomatoes
"Tomatoes"
```

The other three basic kinds of subsetting

Those are the three basic kinds of subsetting, depending on the type of object we use as an index.

But there are also three basic kinds of subsetting on a *different* dimension – the syntax of the command. So far we've just seen one syntax, using `[`.

Subsetting with `[[`

If you want to select a single element, you can subset with double brackets: `[[]]`.

```
food[[1]]
```

```
[1] "Ham"
```

```
food[["ham"]]
```

```
[1] "Ham"
```

There's a subtle difference between `[` and `[[`, can you spot it?

```
food[1]
```

```
  ham
"Ham"
```

```
food[[1]]
```

```
[1] "Ham"
```

Subsetting with `[` keeps the names, subsetting with `[[` doesn't. Actually, it's a bit more than that. Subsetting with `[` really is taking a *subset*. A subset of a named vector is a smaller named vector. Subsetting with `[[` is taking a single *element*. Mathematicians can think of it like this: `[` is to $\subset$ as `[[` is to $\in$.

This gets important when you deal with lists.

```
details <- list(
        name     = "Pete",
        birthday = as.Date("1993-01-01"),
        spouse   = "Miranda"
      )
```

Here, `details[["spouse"]]` will give you the name `"Miranda"`.

```
wife <- details[["spouse"]]
cat("Pete's wife is", wife, ".\n")
```

```
Pete's wife is Miranda .
```

But `details["spouse"]` will give you a list containing one element, named `spouse`. That may not be what you want, because not all R commands accept lists:

```
list_of_wife <- details["spouse"]
cat("This won't work:", list_of_wife)
```

```
This won't work:
```

```
Error in cat("This won't work:", list_of_wife): argument 2 (type '
list') cannot be handled by 'cat'
```

Instead, you can select `list_of_wife[[1]]` to get Pete's first (and only) wife.

Subsetting with `$`

The third kind of subsetting is a close relative of `[[`. It uses `$`.

For example, `details$spouse` is the same as `details[["spouse"]]`.

```
details$spouse
```

```
[1] "Miranda"
```

The two big differences with `[[` are:

1. `$` only takes literal names. So, this works:

```
date <- "birthday"
details[[date]]
```

```
[1] "1993-01-01"
```

But this doesn't, because there's no element literally called `date`:

```
details$date
```

```
NULL
```

2. `$` **only** works with lists. It doesn't work with atomic vectors (vectors of numbers, characters, etc.).

So, this won't work:

```
food$ham
```

```
Error in food$ham: $ operator is invalid for atomic vectors
```

`$` is basically a useful shorthand for `[[`, when you can select an element by name.

Recycling

So far, so easy, but there are already questions. What happens if a logical vector is shorter or longer than what it is indexing?

The answer is that R *recycles* the vector (i.e. repeats it) along the length of the array. So, for example, `food[TRUE]` becomes `food[c(TRUE, TRUE, TRUE)]`, which gives you the whole vector back.

```
food[TRUE]
```

```
      ham       eggs   tomatoes
    "Ham"     "Eggs" "Tomatoes"
```

If the recycling doesn't quite fit, you get a warning:

```
food[c(TRUE, TRUE)] # 2 into 3 doesn't go
```

```
      ham       eggs   tomatoes
    "Ham"     "Eggs" "Tomatoes"
```

Oh! No you don't. I thought you did.

OK, but if your vector is too long, you get a warning:

```
food[c(TRUE, FALSE, TRUE, TRUE)]
```

```
       ham   tomatoes       <NA>
     "Ham" "Tomatoes"         NA
```

Damn.

(Goes to `?Extract`.)

(Nothing there.)

(Goes to R language definition.)

Ah, OK. If the index is longer than the vector being indexed, then the vector is "conceptually extended with NAs". So, above, `c(TRUE, FALSE, TRUE, TRUE)` gave us the first element, not the second element, the third element... and an imaginary "fourth element" which is assumed to be `NA`.

The sound of no hands clapping

While we're on the topic, what happens if you have a `NA` in your index?

```
food[c(TRUE, NA, TRUE)]
```

```
       ham       <NA>   tomatoes
     "Ham"         NA "Tomatoes"
```

```
food[c(1, NA, 3)]
```

```
       ham       <NA>   tomatoes
     "Ham"         NA "Tomatoes"
```

```
food[c("ham", NA, "eggs")]
```

```
   ham   <NA>   eggs
 "Ham"     NA "Eggs"
```

You get a `NA` in the answer.

And if you have a zero-length index, or `NULL`, you get zero elements in the response:

```
food[numeric(0)]
```

```
named character(0)
```

```
food[character(0)]
```

```
named character(0)
```

```
food[logical(0)]
```

```
named character(0)
```

```
food[NULL]
```

```
named character(0)
```

But, what if you have literally **nothing** in the index?

```
food[]
```

```
       ham        eggs    tomatoes
     "Ham"      "Eggs" "Tomatoes"
```

You get everything back. **Wat.**

We're almost over the basics, and we've just got one more thing to know:

Two dimensional subsetting

Some objects in R have two dimensions, like matrices and data frames. So you need two indices to take subsets of them.

Here's the food on sale at the local shop:

```
(food_data <- data.frame(
  name       = food,
  price      = c(2.00, 1.50, 0.80),
  vegan      = c(FALSE, FALSE, TRUE),
  vegetarian = c(FALSE, TRUE, TRUE),
  stringsAsFactors = FALSE,
  row.names = NULL
))
```

```
      name price vegan vegetarian
1      Ham   2.0 FALSE      FALSE
2     Eggs   1.5 FALSE       TRUE
3 Tomatoes   0.8  TRUE       TRUE
```

To get the first two rows and the first four columns of this data frame, we can do:

```
food_data[1:2, 1:4]
```

```
  name price vegan vegetarian
1  Ham   2.0 FALSE      FALSE
2 Eggs   1.5 FALSE       TRUE
```

The first index gives the rows. The second index gives the columns.

As we've included all the columns, there's a shorthand we could use, which is to leave out one index altogether.

```
food_data[1:2, ]
```

```
  name price vegan vegetarian
1  Ham   2.0 FALSE      FALSE
2 Eggs   1.5 FALSE       TRUE
```

Aha! Now you see why it made sense to have "nothing" select everything.

All the kinds of subset we used before still work here. For example, we can get rows 1 and 3 like this:

```
food_data[c(TRUE, FALSE, TRUE), ]
```

```
      name price vegan vegetarian
1      Ham   2.0 FALSE      FALSE
3 Tomatoes   0.8  TRUE       TRUE
```

Or we could get columns like this, by name:

```
food_data[, c("name", "vegan")]
```

```
      name vegan
1      Ham FALSE
2     Eggs FALSE
3 Tomatoes  TRUE
```

A typical pattern is to get columns by name and rows using a logical vector. For example, here's how to get the names and prices of vegetarian food:

```
food_data[food_data$vegetarian, c("name", "price")]
```

```
      name price
2     Eggs   1.5
3 Tomatoes   0.8
```

That seems clear enough... only, what is `food_data$vegetarian` ? It must be giving the logical index that selects only vegetarian food — but didn't I say that `$` only worked for lists?

I did and it does. The reason this works is that a data frame is secretly a list. It's just a list of columns. So, `food_data$vegetarian` selects one element — i.e. one column. Here's some more examples:

```
food_data$name # The 'name' column is a character vector
```

```
[1] "Ham"      "Eggs"     "Tomatoes"
```

```
food_data$price # The 'price' column is a numeric vector
```

```
[1] 2.0 1.5 0.8
```

And here's how to get all foods with a price of over 1.40:

```
food_data[food_data$price > 1.40, ]
```

```
  name price vegan vegetarian
1  Ham   2.0 FALSE      FALSE
2 Eggs   1.5 FALSE       TRUE
```

As you might expect, the `[[` subsetting works just the same way as `$` . You can use it to select a single element of the data frame – i.e. a single column. Unlike `$` , you can use it

with computed names as well as literal names:

```
my_diet_preference <- "vegan"
ok_to_eat <- food_data[[my_diet_preference]]
food_data[ok_to_eat, ]
```

```
      name price vegan vegetarian
3 Tomatoes   0.8  TRUE       TRUE
```

We could have written those last two lines as one line:

```
food_data[ food_data[[my_diet_preference]], ]
```

but all those `[` characters together can make your eyes start to cross.

You can also select a single column using `[` . This is useful when you also want to select a subset of rows:

```
# Names of vegan food:
food_data[food_data$vegetarian, "name"]
```

```
[1] "Eggs"     "Tomatoes"
```

Advanced subsetting

Now, for some stuff you might not know.

Don't drop that!

If a data frame is a list, and `[` selects subsets of lists, then you might have expected `food_data[, "name"]` to select a subset of that data frame – i.e. a smaller data frame with one column. That would have been consistent, and it might have saved a lot of pain for R developers over the years... but as you can see above, it didn't happen. When you selected `food_data[, "name"]` , you just got a vector, not a data frame with one column.

To repeat: if you select a single column of a data frame using `[` , you get just a vector of data. If you select two or more columns, you get the whole data frame.

This is a powerful foot shotgun.

Maybe we write a complex computation to select certain columns:

```
frobnitz_the_zacular <- function () 2:3

calculated_columns <- frobnitz_the_zacular()
food_to_buy <- food_data[, calculated_columns]
nrow(food_to_buy)
```

```
[1] 3
```

Later on, we look through our complex code and realise we can reverse the Hyperlight Drive:

```
frobnitz_the_zacular <- function () 2
```

Oh dear.

```
calculated_columns <- frobnitz_the_zacular()
food_to_buy <- food_data[, calculated_columns]
nrow(food_to_buy)
```

```
NULL
```

The way to avoid this is the special argument `drop = FALSE`. You can put this after your rows and columns in the subset:

```
calculated_columns <- frobnitz_the_zacular()
food_to_buy <- food_data[, calculated_columns, drop = FALSE]
nrow(food_to_buy)
```

```
[1] 3
```

You will forget this. And then, you will regret it.

OH NO! OWL BEARS!

There's another gotcha with `$` subsetting. I'll just feed my pet owl:

```
cat("My pet owl says to-whit", pets$owl)
```

```
My pet owl says to-whit TO-WRAAAARGH!
```

Oh... that wasn't an owl! Unfortunately, my owl escaped from the list, and I tried to feed an owlbear:

```
str(pets)
```

```
List of 1
 $ owlbear: chr "TO-WRAAAARGH!"
```

In other words, `$` matches using substrings, if it can't find an exact match. So you may not always get what you expect.

Minus numbers

Just as you use positive integers to *select* elements, you can use negative numbers in indices to *remove* elements.

```
food[-2]
```

```
       ham   tomatoes
     "Ham" "Tomatoes"
```

```
food[c(-1, -3)]
```

```
  eggs
"Eggs"
```

Of course, this works for data frames too. (I won't keep saying that from now on.)

```
# Everything but the first column:
food_data[, -1]
```

```
  price vegan vegetarian
1   2.0 FALSE      FALSE
2   1.5 FALSE       TRUE
3   0.8  TRUE       TRUE
```

If you mix positive and negative numbers, you get an error. No, really.

```
food[c(-2, 2)]
```

```
Error in food[c(-2, 2)]: only 0's may be mixed with negative subsc
ripts
```

What about 0? It turns out that 0 just gets ignored.

```
food[c(0, 1)]
```

```
  ham
"Ham"
```

Matrix indexing

So far, we have only picked out a "square" of data from a data frame. That is, we can subset rows and columns. But what if we want to have, say, the element `[1, 2]`, the element `[3, 3]` and the element `[3, 4]`?

You can do this by indexing with a 2-column matrix. The first column gives the row numbers to pick from the original data, the second column gives the column numbers. Results will be coerced into a vector:

```
elements <- matrix(
        c(1, 2,
          3, 3,
          3, 4),
        nrow  = 3,
        byrow = TRUE
      )

food_data[elements]
```

```
[1] "2.0"   " TRUE" " TRUE"
```

However many dimensions you want

I've assumed that one-dimensional objects get one dimensional indexing, and two-dimensional objects get two dimensional indexing. But really, all objects in R are one dimensional. They just pretend to have more dimensions if you ask them nicely.

If you want to, you can just use one-dimensional indexing. We saw this already with `food_data$vegan` and `food_data[["vegan"]]`. This treated our data frame as a one dimensional list of columns. You can do this with `[` as well:

```
food_data[1:2]
```

```
      name price
1      Ham   2.0
2     Eggs   1.5
3 Tomatoes   0.8
```

This does just the same thing as `food_data[, 1:2]`. On the downside, it's less clear that you are selecting columns not rows.

On the very very up side, the following are *not* equivalent.

```
food_data[, "name"] # a vector
```

```
[1] "Ham"      "Eggs"     "Tomatoes"
```

```
food_data["name"] # a one-column data frame!
```

```
      name
1      Ham
2     Eggs
3 Tomatoes
```

In other words, `food_data["name"]` *doesn't* drop to a vector. As it is also easier to read than `food_data[, "name", drop = FALSE]`, this is a useful trick for defensive code.

But there's more.

Flattening out

Not everything with 2 dimensions is a data frame. There are also matrices. We can use 1 dimensional indexing on these too.

Here's the `euro.cross` data of conversion rates between various European currencies when the € was introduced:

```
euro.cross <- round(euro.cross, 2)
euro.cross[1:3, 1:3]
```

```
     ATS   BEF  DEM
ATS 1.00  2.93 0.14
BEF 0.34  1.00 0.05
DEM 7.04 20.63 1.00
```

```
rownames(euro.cross)
```

```
 [1] "ATS" "BEF" "DEM" "ESP" "FIM" "FRF" "IEP" "ITL" "LUF" "NLG" "
PTE"
```

```
colnames(euro.cross)
```

```
 [1] "ATS" "BEF" "DEM" "ESP" "FIM" "FRF" "IEP" "ITL" "LUF" "NLG" "
PTE"
```

We know how to pick out denmark's currency rates – just do `euro.cross["DEM", ]`. What if we want to pick out the currencies that are within 80% of each other's value?

```
euro.cross[euro.cross < 10/8 & euro.cross > 8/10]
```

```
 [1] 1.00 1.00 1.00 1.00 0.89 1.00 0.83 1.00 0.91 1.10 1.00 1.00
1.00 1.00
[15] 1.00 1.13 1.00 1.20 1.00
```

Here is how this works:

1. The index is a logical matrix.
2. Subsetting treats this as a logical vector.
3. That returns a vector of elements of `euro.cross`.
4. As the subsetting is one dimensional, the dimensions of `euro.cross` are ignored, and the result is just a one-dimensional vector.

This may not seem so useful – how do we know which elements we've got? But it is good if we want to change some elements. For example, we might want to run a counterfactual and look at Eurozone currency rates if these currencies had entered the Euro at parity with each other. Then our data would be:

```
euro.cross[euro.cross < 10/8 & euro.cross > 8/10] <- 1
euro.cross
```

```
      ATS   BEF  DEM    ESP  FIM  FRF  IEP     ITL   LUF  NLG    P
TE
ATS  1.00  2.93 0.14  12.09 0.43 0.48 0.06  140.71  2.93 0.16  14.
57
BEF  0.34  1.00 0.05   4.12 0.15 0.16 0.02   48.00  1.00 0.05   4.
97
DEM  7.04 20.63 1.00  85.07 3.04 3.35 0.40  990.00 20.63 1.00 102.
50
ESP  0.08  0.24 0.01   1.00 0.04 0.04 0.00   11.64  0.24 0.01   1.
00
FIM  2.31  6.78 0.33  27.98 1.00 1.00 0.13  325.66  6.78 0.37  33.
72
FRF  2.10  6.15 0.30  25.37 1.00 1.00 0.12  295.18  6.15 0.34  30.
56
IEP 17.47 51.22 2.48 211.27 7.55 8.33 1.00 2458.56 51.22 2.80 254.
56
ITL  0.01  0.02 0.00   0.09 0.00 0.00 0.00    1.00  0.02 0.00   0.
10
LUF  0.34  1.00 0.05   4.12 0.15 0.16 0.02   48.00  1.00 0.05   4.
97
NLG  6.24 18.31 1.00  75.50 2.70 2.98 0.36  878.64 18.31 1.00  90.
97
PTE  0.07  0.20 0.01   1.00 0.03 0.03 0.00    9.66  0.20 0.01   1.
00
```

How does it mean to treat a 2D object as a 1D object? That depends.

A data frame is just a list of columns, so indexing in one dimension just gives you columns. For example, `food_data[1]` is the first column of food data.

A matrix isn't a list of columns; it's a vector with a `dim()` attribute. The vector indexing goes down the columns. Here's an illustration:

```
rc <- paste(rep(1:3, 3), rep(1:3, each = 3), sep = ",")
m <- matrix(rc, 3, 3)
```

```
ht
```

A 3x3 matrix

1,1	1,2	1,3

2,1	2,2	2,3
3,1	3,2	...

And here's how it gets unwound by matrix indexing:

```
ht2
```

The matrix when indexed by a single vector

1 (1,1)
2 (2,1)
3 (3,1)
4 (1,2)
5 (2,2)
6 (3,2)
7 (1,3)
8 (2,3)
...

So, for example, element 5 is the yellow one – row 2 column 2.

The ability to ignore dimensions is great when you are using functions that don't preserve dimensions themselves. For example, suppose I want to change the text of this matrix by adding some labels:

```
paste("Row, column: ", m)
```

```
[1] "Row, column:  1,1" "Row, column:  2,1" "Row, column:  3,1"
[4] "Row, column:  1,2" "Row, column:  2,2" "Row, column:  3,2"
[7] "Row, column:  1,3" "Row, column:  2,3" "Row, column:  3,3"
```

Unfortunately, `paste` doesn't preserve dimensions, so if I did

```
m <- paste("Row, column:", m)
```

I'd have a vector, not the matrix I wanted.

But the result is what I need. It just needs its dimensions back. I could either do `dim(m) <- c(3, 3)`, or, to avoid any errors with the dimensions wrong, I can put my data back into the matrix shape with the "empty subset" trick:

```
m[] <- paste("Row, column:", m)
m[]
```

```
     [,1]                [,2]                [,3]
[1,] "Row, column: 1,1" "Row, column: 1,2" "Row, column: 1,3"
[2,] "Row, column: 2,1" "Row, column: 2,2" "Row, column: 2,3"
[3,] "Row, column: 3,1" "Row, column: 3,2" "Row, column: 3,3"
```

Using `m[] <-` assigns into every element of `m`. That is different from `m<-`, which overwrites the old object with a whole new one.

In fact, you can mix 2-dimensional subsetting with one dimensional subsetting, at your convenience. For example, suppose I only want the "row, column" labels at the top of every column:

```
m <- matrix(rc, 3, 3)
# In 1D space:
with_labels <- paste("Row, column:", m)
# In 2D space:
dim(with_labels) <- dim(m)
m[1, ] <- with_labels[1, ]
m
```

```
     [,1]                [,2]                [,3]
[1,] "Row, column: 1,1" "Row, column: 1,2" "Row, column: 1,3"
[2,] "2,1"              "2,2"               "2,3"
[3,] "3,1"              "3,2"               "3,3"
```

I used this trick a lot recently, when I rewrote the huxtable (https://hughjonesd.github.io/huxtable) package, which produces LaTeX and HTML tables. (Huxtable produced the pictures above.) Obviously table data comes in 2D form, but sometimes you want to work on a subset of it, treating the subset as just a vector, then just assign back into the matrix. The pattern is:

```
vec_subset <- matrix_data[rows, cols]
# if you need to explicitly remove the dimensions, you can do:
vec_subset <- c(vec_subset)

# Now do a bunch of vector operations:
vec_subset <- manipulate(vec_subset)

# Assign back into the matrix
matrix_data[rows, cols] <- vec_subset
```

Vector operations are fast, so this can be a nice way to speed up code.

Double brackets, many dimensions

You normally only see `[[` used with one index, to select from a list. But you can use it with multiple dimensions just like `[`. It always only returns one element:

```
food_data[[1, "name"]]
```

```
[1] "Ham"
```

So, if you know your code should only want one element, this is a safe way of asking for it.

Going deep

I never even knew about this until this year, and have never used it.

Lists are good for recursive data structures:

```
details <- list(
      name = "Henry",
      birthday = as.Date("1491-07-28"),
      wives = list(
          Catherine = list(languages = c("Spanish", "English", chi
ldren = "Mary")),
          Anne = list(motto = "Grogne qui grogne...", children = "
Elizabeth"),
          Jane = list(children = "Edward"),
          Ann  = list(from = "Flanders"),
          Katherine = list(bad_habits = "flirtatious"),
          Catherine = list(religion = "Protestant")
        )
      )
```

What if we want to dive into those structures? Say, finding the name of Henry's second wife's child?

You can deep dive into a recursive (list-like) object with `[[` and a vector index:

```
details[[c("wives", "Anne", "children")]]
```

```
[1] "Elizabeth"
```

Notice that this only works with recursive objects. If you subset an atomic vector using `[[` and a vector index, you'll get an error.

Summary

Here's a summary of how indexing works:

Vector	Index	`[`	`[[`
Atomic	numeric	Subset selected by position	Single element selected by position
	character	Subset selected by names	Single element selected by name
	logical	Subset where index is `TRUE`	--

Recursive	numeric	Subset selected by position	Single element selected by position
	character	Subset selected by names	Single element by name (use vector for deep indexing)
	logical	Subset where index is `TRUE`	--

That's all! Or at least, it's all I know. Maybe in 15 years, I'll discover some more tricks. Meanwhile, good luck, and don't behead yourself.