

## Subsetting in R

“Subsetting” refers to the act of getting a “subset” of a list, vector, or other structure in R.

The material is somewhat dry, so feel free to take breaks, but it’s important to master it – subsetting is absolutely essential to doing any substantial amount of work in R.

Wickham writes:

Subsetting is a natural complement to `str()`. `str()` shows you the structure of any object, and subsetting allows you to pull out the pieces that you’re interested in.

Keep the `str()` function in mind as you work through this lesson.

### Simple single-bracket subsetting

The simplest form of subsetting uses the single brackets `[` and `]`. We’ll cover how this works with a variety of data types. Sometimes, we’ll refer to the vector used to subset a different vector as the *index vector*. (For example, in `x[values]`, `values` is the index vector.)

There are three main ways to subset an unnamed vector—figure out how they work by playing around with `x = 1:5`.

- Subsetting with positive integers: `x[c(3,1)]`
- Subsetting with negative integers: `x[-c(3,1)]`, `x[c(-3,-1)]`
- Subsetting with logical vectors: `x[c(TRUE, FALSE, TRUE, FALSE, TRUE)]`

**Exercise.** What happens when you subset with multiple copies of the same positive integer?

**Exercise.** What happens when you subset with numbers that are not whole numbers?

**Exercise.** What happens when you subset with both positive and negative integers?

**Exercise.** What happens when you subset with a logical vector shorter than the vector you’re subsetting? Try with short logical vectors of length 1 and 2.

**Exercise.** What happens when there are some `NA` values in the index vector?

Moreover, you can pass in *character vectors* as the index vector to subset based on *names*.

**Exercise.** What happens if you try to subset by name but one of the values you pass in isn't a valid name?

**Exercise.** Use `[]` to play around with subsetting lists. What *type* of object do you get back?

Recall that data frames are simply complex, two-dimensional lists. If you subset with a *single vector*, data frames behave identically to lists. However, you can simultaneously subset both dimensions by passing in *two* vectors. It's easiest to demonstrate:

```
> df = data.frame(matrix(1:100, nrow=10, ncol=10))
> df
   X1 X2 X3 X4 X5 X6 X7 X8 X9 X10
1   1  2  3  4  5  6  7  8  9 10
2  11 12 13 14 15 16 17 18 19 20
3  21 22 23 24 25 26 27 28 29 30
4  31 32 33 34 35 36 37 38 39 40
5  41 42 43 44 45 46 47 48 49 50
6  51 52 53 54 55 56 57 58 59 60
7  61 62 63 64 65 66 67 68 69 70
8  71 72 73 74 75 76 77 78 79 80
9  81 82 83 84 85 86 87 88 89 90
10 91 92 93 94 95 96 97 98 99 100
> df[2:4, 3:6]
   X3 X4 X5 X6
2 22 32 42 52
3 23 33 43 53
4 24 34 44 54
```

**Exercise.** Does subsetting a data frame with a single vector select rows or columns?

**Exercise.** What happens when you pass in nothing for one of the two vectors, like with `df[1:2,]` or `df[,5:6]`?

**Exercise.** When subsetting with two vectors, can you pass in a vector of column names?

**Exercise.** What's the difference between (1) subsetting a single column by passing in a single number as the index vector versus (2) subsetting a single column by passing in nothing for the first index vector and a single number for the second index vector?<sup>1</sup>

## Advanced subsetting

Wickham writes:

---

<sup>1</sup>You'll very soon learn an easier way of grabbing the vector directly.

There are two other subsetting operators: `[[` and `$`. `[[` is similar to `[`, except it can only return a single value and it allows you to pull pieces out of a list. `$` is a useful shorthand for `[[` combined with character subsetting.

You need `[[` when working with lists. This is because when `[` is applied to a list it always returns a list: it never gives you the contents of the list. To get the contents, you need `[[`:

“If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6.”

– @RLangTip

Because it can return only a single value, you must use `[[` with either a single positive integer or a string.

This works straightforwardly. Using the same 10-by-10 data frame from earlier, you can grab the contents of the 5th column with `df[[5]]`, `df[["V5"]]`, or `df$V5`. The `$` operator is nearly identical to the `[[` operator.<sup>2</sup>

There are nuances to the behavior of all of these different operators, but for they most part they aren't important – we've covered all the essential parts already. If you want to read more about the details, then consult section 3.2.1 in Wickham's *Advanced R*.

## Supplementary exercises

You can modify objects by subsetting them and then using a standard assignment operator (`=`) to assign values to the subsets.

In the following, `mtcars` refers to a dataset that's loaded by default.

Finally, columns of a data frame can be removed by assigning `NULL` to them.

**Exercise.** With a single subset assignment command, change `x = 1:5` to be equivalent to `c(10, 11, 3, 4, 5)`.

**Exercise.** With a single subset assignment command, change `x = 1:10` to be equivalent to `c(1, 100, 3, 100, 5, 10, 7, 100, 9, 100)`. (*Hint:* You can subset with a logical vector.)<sup>3</sup>

**Advanced R, 3.1.7.2.** Why does `x = 1:5; x[NA]` yield five missing values? (*Hint:* How is it different from `NA_real_`?)

---

<sup>2</sup>There's one minor exception. `x$y` is actually equivalent to `x[["y", exact = FALSE]]`, so `$` can partially match names (starting from the beginning of the string). For example, if `df` has a column named `"column"`, then `df$c` will return the output of `df$column`, assuming that no other columns in `df` have a name beginning with `"c"`.

<sup>3</sup>A good way to do this is with `x[x %% 2 == 0] = rep(100, length(x[x %% 2 == 0]))`. We pass in the value of `length(...)` instead of 5 directly to improve the robustness of our code – our manual calculation of the value 5 could be incorrect.

**Advanced R, 3.1.7.4.** Why does `mtcars[1:20]` return an error? How does it differ from the similar `mtcars[1:20, ]`?

**Advanced R, 3.1.7.6.** What does `df[is.na(df)] = 0` do? How does it work?

**Exercise.** Let `x = c("a", "b", "a", "a", "b", "x", "b", "a")`. Construct a named vector called `fruits` such that the output of `fruits[x]` is equal to `c("apple", "banana", "apple", "apple", "banana", NA, "banana", "apple")`.

**Exercise.** Using the `order()` function, write a function to alphabetize the columns of a data frame by their names.

**Exercise.** Using the `sample()` function, write a function that takes a data frame as input and returns it with the order of its columns randomly permuted. After that, add a logical (boolean) flag to the function's parameters called `rows` defaulting to `FALSE` that permutes the rows as well if set to `TRUE`. (*I.e.*, calling `f(df)` would be equivalent to calling `f(df, rows=FALSE)` but `f(df, rows=TRUE)` would permute rows as well as columns.)

**Exercise.** Write a function that takes a data frame `df` and an integer `k` as input and returns `k` random columns of `df`, *sampled with replacement*.

**Exercise.** Write a function that takes a data frame `df` and an integer `m` as input and returns a random sample of `m` continuous rows of `df` as the output. (By continuous, we mean that you would return row `i`, row `i+1`, ... all the way to row `i+m-1` for some `i`.)

**Exercise.** Write a function that takes a data frame `df` and a string `colname` as input and returns a dataframe without any columns that have name equal to the value of `colname`. (*Hint:* Don't forget about the edge case where multiple columns have identical names.)