Subsetting

PS239T

06 February, 2019

Subsetting

When working with data, you'll need to subset objects early and often. Luckily, R's subsetting operators are powerful and fast. Mastery of subsetting allows you to succinctly express complex operations in a way that few other languages can match. Subsetting is hard to learn because you need to master a number of interrelated concepts:

- The three subsetting operators, `[`, `[[`, and `$`.
- Important differences in behaviour for different objects (e.g., vectors, lists, factors, matrices, and data frames).
- The use of subsetting in conjunction with assignment.

This unit helps you master subsetting by starting with the simplest type of subsetting: subsetting an atomic vector with `[`. It then gradually extends your knowledge, first to more complicated data types (like dataframes and lists), and then to the other subsetting operators, `[[` and `$`. You'll then learn how subsetting and assignment can be combined to modify parts of an object, and, finally, you'll see a large number of useful applications.

Outline

1. Data types starts by teaching you about `[`. You'll start by learning the four ways to subset atomic vectors. You'll then learn how those four methods act when used to subset lists, matrices, and data frames.
2. Subsetting operators expands your knowledge of subsetting operators to include `[[` and `$`, focussing on the important principles of simplifying vs. preserving.
3. In Subsetting and assignment you'll learn the art of subassignment, combining subsetting and assignment to modify parts of an object.
4. Applications leads you through important, but not obvious, applications of subsetting to solve problems that you often encounter in a data analysis, using the tools above.
5. Bonus Items show you some additional details of how to work with your data and make your code more efficient, including by distinguishing when you want to simplify vs. preserve your data.

1. Data types

It's easiest to learn how subsetting works for atomic vectors, and then how it generalises to higher dimensions and other more complicated objects. We'll start with `[`, the most commonly used operator. Subsetting operators will cover `[[` and `$`, the two other main subsetting operators.

1a. Atomic vectors

Let's explore the different types of subsetting with a simple vector, `x`.

```
x <- c(2.1, 4.2, 3.3, 5.4)
```

Note that the number after the decimal point gives the original position in the vector.

NB: In R, positions start at 1, unlike Python, which starts at 0. Fun!**

There are five things that you can use to subset a vector:

1. Positive integers return elements at the specified positions:

```
x <- c(2.1, 4.2, 3.3, 5.4)
x
```

```
## [1] 2.1 4.2 3.3 5.4
```

```
x[1]
```

```
## [1] 2.1
```

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

```
## [1] 3.3 2.1
```

```
# `order(x)` gives the positions of smallest to largest values.
order(x)
```

```
## [1] 1 3 2 4
```

```
x[order(x)]
```

```
## [1] 2.1 3.3 4.2 5.4
```

```
x[c(1, 3, 2, 4)]
```

```
## [1] 2.1 3.3 4.2 5.4
```

```
# Duplicated indices yield duplicated values
x[c(1, 1)]
```

```
## [1] 2.1 2.1
```

2. Negative integers omit elements at the specified positions:

```
x <- c(2.1, 4.2, 3.3, 5.4)
x[-1]
```

```
## [1] 4.2 3.3 5.4
```

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

```
## [1] 4.2 5.4
```

You can't mix positive and negative integers in a single subset:

```
x <- c(2.1, 4.2, 3.3, 5.4)
x[c(-1, 2)]
```

```
## Error in x[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

3. Logical vectors select elements where the corresponding logical value is `TRUE`.

```
x <- c(2.1, 4.2, 3.3, 5.4)

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

```
## [1] 2.1 4.2
```

This is probably the most useful type of subsetting because you write the expression that creates the logical vector

```
x <- c(2.1, 4.2, 3.3, 5.4)

# this returns a logical vector
x > 3
```

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

```
x
```

```
## [1] 2.1 4.2 3.3 5.4
```

```
# use a conditional statement to create an implicit logical vector
x[x > 3]
```

```
## [1] 4.2 3.3 5.4
```

You can combine conditional statements with `&` (and), `|` (or), and `!` (not)

```
x <- c(2.1, 4.2, 3.3, 5.4)

# combing two conditional statements with &
x > 3 & x < 5
```

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

```
x[x > 3 & x < 5]
```

```
## [1] 4.2 3.3
```

```
# combing two conditional statements with |
x < 3 | x > 5
```

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

```
x[x < 3 | x > 5]
```

```
## [1] 2.1 5.4
```

```
# combining conditional statements with !
!x > 5
```

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

```
x[!x > 5]
```

```
## [1] 2.1 4.2 3.3
```

Another way to generate implicit conditional statements is using the `%in%` operator, which works like the `in` keywords in Python.

```
# generate implicit logical vectors through the %in% operator
x %in% c(3.3, 4.2)
```

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

```
x
```

```
## [1] 2.1 4.2 3.3 5.4
```

```
x[x %in% c(3.3, 4.2)]
```

```
## [1] 4.2 3.3
```

4. Character vectors to return elements with matching names. This only works if the vector is named.

```
x <- c(2.1, 4.2, 3.3, 5.4)

# apply names
names(x) <- c("a", "b", "c", "d")
x
```

```
##   a   b   c   d
## 2.1 4.2 3.3 5.4
```

```
# subset using names
x[c("d", "c", "a")]
```

```
##   d   c   a
## 5.4 3.3 2.1
```

```
# Like integer indices, you can repeat indices
x[c("a", "a", "a")]
```

```
##   a   a   a
## 2.1 2.1 2.1
```

```
# Careful! names are always matched exactly
x <- c(abc = 1, def = 2)
x
```

```
## abc def
##   1   2
```

```
x[c("a", "d")]
```

```
## <NA> <NA>
##   NA   NA
```

Exercise

Subset `country.vector` below to return every value EXCEPT "Canada" and "Brazil"

```
country.vector<-c("Afghanistan", "Canada", "Sierra Leone", "Denmark", "Japan", "Brazil")

# Do it using positive integers

country.vector[c(______)]

# Do it using negative integers

country.vector[-c(_____)]

# Do it using a logical vector

country.vector[c(_______)]

# Do it using a conditional statement (and an implicit logical vector)
```

```
country.vector[!________ %in% c(________)]
```

1b. Lists

Subsetting a list works in the same way as subsetting an atomic vector. Using `[` will always return a list; `[[` and `$`, as described below, let you pull out the components of the list.

```
l <- list('a' = 1, 'b' = 2)
l
```

```
## $a
## [1] 1
##
## $b
## [1] 2
```

```
l[1]
```

```
## $a
## [1] 1
```

```
l[[1]]
```

```
## [1] 1
```

```
l['a']
```

```
## $a
## [1] 1
```

1c. Matrices

The most common way of subsetting matrices (2d) is a simple generalisation of 1d subsetting: you supply a 1d index for each dimension, separated by a comma. Blank subsetting is now useful because it lets you keep all rows or all columns.

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- c("A", "B", "C")
a
```

```
##      A B C
## [1,] 1 4 7
## [2,] 2 5 8
## [3,] 3 6 9
```

```
# rows come first, then columns
a[c(1, 2), ]
```

```
##      A B C
## [1,] 1 4 7
## [2,] 2 5 8
```

```
a[c(T, F, T), c("B", "A")]
```

```
##      B A
## [1,] 4 1
## [2,] 6 3
```

```
a[0, -2]
```

```
##      A C
```

```
a[c(1,2) , -2]
```

```
##      A C
## [1,] 1 7
## [2,] 2 8
```

1d. Data frames

Data from data frames can be addressed like matrices (with row and column indicators separated by a comma).

```
df <- data.frame(x = 4:6, y = 3:1, z = letters[1:3])
df
```

```
##   x y z
## 1 4 3 a
## 2 5 2 b
## 3 6 1 c
```

```
# return only the rows where x == 6
df[df$x == 6, ]
```

```
##   x y z
## 3 6 1 c
```

```
# return the first and third row
df[c(1, 3), ]
```

```
##   x y z
## 1 4 3 a
## 3 6 1 c
```

```
# return the first and third row, and the first and second column
df[c(1, 3), c(1,2)]
```

```
##   x y
## 1 4 3
## 3 6 1
```

Data frames possess the characteristics of both lists and matrices: if you subset with a single vector, they behave like lists, and return only the columns.

```
# There are two ways to select columns from a data frame
# Like a list:
df[c("x", "z")]
```

```
##   x z
## 1 4 a
## 2 5 b
## 3 6 c
```

```
# Like a matrix
df[, c("x", "z")]
```

```
##   x z
## 1 4 a
```

```
## 2 5 b
## 3 6 c
```

But there's an important difference when you select a single column: matrix subsetting simplifies by default, list subsetting does not.

```
(df["x"])
```

```
##   x
## 1 4
## 2 5
## 3 6
```

```
class((df["x"]))
```

```
## [1] "data.frame"
```

```
(df[, "x"])
```

```
## [1] 4 5 6
```

```
class((df[, "x"]))
```

```
## [1] "integer"
```

See the bottom section on Simplying and Preserving to know more

Exercises

1. Fix each of the following common data frame subsetting errors:

```
# check out what we're dealing with
mtcars

# fix
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]

# answers
mtcars[mtcars$cyl == 4, ]
mtcars[-c(1:4), ]
mtcars[mtcars$cyl <= 5,]
mtcars[mtcars$cyl == 4 | mtcars$cyl == 6, ]
```

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

2. Subsetting operators

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.

2a. `[[`

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 data frames are lists of columns, you can use `[[` to extract a column from data frames:

```
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
# these two are equivalent
mtcars[[1]]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

```
mtcars[ ,1]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
```

```
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

```
# which differs from this:
mtcars[1]
```

```
##                      mpg
## Mazda RX4           21.0
## Mazda RX4 Wag       21.0
## Datsun 710          22.8
## Hornet 4 Drive      21.4
## Hornet Sportabout   18.7
## Valiant             18.1
## Duster 360          14.3
## Merc 240D           24.4
## Merc 230            22.8
## Merc 280            19.2
## Merc 280C           17.8
## Merc 450SE          16.4
## Merc 450SL          17.3
## Merc 450SLC         15.2
## Cadillac Fleetwood  10.4
## Lincoln Continental 10.4
## Chrysler Imperial   14.7
## Fiat 128            32.4
## Honda Civic         30.4
## Toyota Corolla      33.9
## Toyota Corona       21.5
## Dodge Challenger    15.5
## AMC Javelin         15.2
## Camaro Z28          13.3
## Pontiac Firebird    19.2
## Fiat X1-9           27.3
## Porsche 914-2       26.0
## Lotus Europa        30.4
## Ford Pantera L      15.8
## Ferrari Dino        19.7
## Maserati Bora       15.0
## Volvo 142E          21.4
```

2b. `$`

`$` is a shorthand operator, where `x$y` is equivalent to `x[["y", exact = FALSE]]`. It's often used to access variables in a data frame:

```
# these two are equivalent
mtcars[["cyl"]]
```

```
##  [1] 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
```

```
mtcars$cyl
```

```
##  [1] 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
```

One common mistake with `$` is to try and use it when you have the name of a column stored in a variable:

```
var <- "cyl"
# Doesn't work - mtcars$var translated to mtcars[["var"]]
mtcars$var
```

```
## NULL
```

```
# Instead use [[
mtcars[[var]]
```

```
##  [1] 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
```

Exercises

1. Take a look at the linear model below:

```
mod <- lm(mpg ~ wt, data = mtcars)
summary(mod)
```

```
##
## Call:
## lm(formula = mpg ~ wt, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.5432 -2.3647 -0.1252  1.4096  6.8727
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  37.2851     1.8776  19.858  < 2e-16 ***
## wt           -5.3445     0.5591  -9.559 1.29e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.046 on 30 degrees of freedom
## Multiple R-squared:  0.7528, Adjusted R-squared:  0.7446
## F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

Extract the R squared from the model summary.

```
mod.sum <- summary(mod)
mod.sum$______ # Fill in here
```

```
## Error: <text>:2:9: unexpected input
## 1: mod.sum <- summary(mod)
## 2: mod.sum$_
##            ^
```

3. Subassignment

All subsetting operators can be combined with assignment to modify selected values of the input vector.

```
x <- 1:5
x
```

```
## [1] 1 2 3 4 5
```

```r
x[c(1, 2)] <- 2:3
x
```

```
## [1] 2 3 3 4 5
```

```r
# The length of the LHS needs to match the RHS!
x[-1] <- 4:1
x
```

```
## [1] 2 4 3 2 1
```

```r
x[1] <- 4:1
```

```
## Warning in x[1] <- 4:1: number of items to replace is not a multiple of
## replacement length
```

```r
# This is mostly useful when conditionally modifying vectors
df <- data.frame(a = c(1, 10, NA))
df
```

```
##    a
## 1  1
## 2 10
## 3 NA
```

```r
df$a[df$a < 5] <- 0
df
```

```
##    a
## 1  0
## 2 10
## 3 NA
```

4. Applications

The basic principles described above give rise to a wide variety of useful applications. Some of the most important are described below. Many of these basic techniques are wrapped up into more concise functions (e.g., `subset()`, `merge()`, `plyr::arrange()`), but it is useful to understand how they are implemented with basic subsetting. This will allow you to adapt to new situations that are not dealt with by existing functions.

4a. Ordering Columns

Consider we have this data frame:

```r
df <- data.frame(
  Country = c("Iraq", "China", "Mexico", "Russia", "United Kingdom"),
  Region = c("Middle East", "Asia", "North America", "Eastern Europe", "Western Europe"),
  Language = c("Arabic", "Mandarin", "Spanish", "Russian", "English")
)
df
```

```
##          Country         Region Language
## 1           Iraq    Middle East   Arabic
## 2          China           Asia Mandarin
## 3         Mexico  North America  Spanish
## 4         Russia Eastern Europe  Russian
```

```
## 5 United Kingdom Western Europe  English
```

What if we wanted to reorder the columns so that `Region` is first? We can do so using subsetting with the names (or number) of the columns:

```
df <- data.frame(
  Country = c("Iraq", "China", "Mexico", "Russia", "United Kingdom"),
  Region = c("Middle East", "Asia", "North America", "Eastern Europe", "Western Europe"),
  Language = c("Arabic", "Mandarin", "Spanish", "Russian", "English")
)

# reorder columns using names
names(df)
```

```
## [1] "Country"  "Region"   "Language"
```

```
df1 <- df[, c("Region", "Country", "Language")]
df1
```

```
##           Region        Country Language
## 1    Middle East           Iraq   Arabic
## 2           Asia          China Mandarin
## 3  North America         Mexico  Spanish
## 4 Eastern Europe         Russia  Russian
## 5 Western Europe United Kingdom  English
```

```
# reorder columns using indices
names(df)
```

```
## [1] "Country"  "Region"   "Language"
```

```
df1 <- df[, c(2,1,3)]
df1
```

```
##           Region        Country Language
## 1    Middle East           Iraq   Arabic
## 2           Asia          China Mandarin
## 3  North America         Mexico  Spanish
## 4 Eastern Europe         Russia  Russian
## 5 Western Europe United Kingdom  English
```

One helpul function is the `order` function, which takes a vector as input and returns an integer vector describing how the subsetted vector should be ordered:

```
x <- c("b", "c", "a")
order(x)
```

```
## [1] 3 1 2
```

```
x[order(x)]
```

```
## [1] "a" "b" "c"
```

Knowing this, we can use `order` to reorder our columns by alphabetical order.

4b. Removing (or keeping) columns from data frames

There are two ways to remove columns from a data frame. You can set individual columns to `NULL`:

```r
df <- data.frame(
  Country = c("Iraq", "China", "Mexico", "Russia", "United Kingdom"),
  Region = c("Middle East", "Asia", "North America", "Eastern Europe", "Western Europe"),
  Language = c("Arabic", "Mandarin", "Spanish", "Russian", "English")
)

df$Language <- NULL
```

Or you can subset to return only the columns you want:

```r
df <- data.frame(
  Country = c("Iraq", "China", "Mexico", "Russia", "United Kingdom"),
  Region = c("Middle East", "Asia", "North America", "Eastern Europe", "Western Europe"),
  Language = c("Arabic", "Mandarin", "Spanish", "Russian", "English")
)

df1 <- df[, c("Country", "Region")]
df1
```

```
##          Country         Region
## 1           Iraq    Middle East
## 2          China           Asia
## 3         Mexico  North America
## 4         Russia Eastern Europe
## 5 United Kingdom Western Europe
```

```r
# using negative integers
df2 <- df[, -3]
df2
```

```
##          Country         Region
## 1           Iraq    Middle East
## 2          China           Asia
## 3         Mexico  North America
## 4         Russia Eastern Europe
## 5 United Kingdom Western Europe
```

4c. Selecting rows based on a condition (logical subsetting)

Because it allows you to easily combine conditions from multiple columns, logical subsetting is probably the most commonly used technique for extracting rows out of a data frame.

```r
mtcars[mtcars$gear == 5, ]
```

```
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
```

```r
mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
```

```
##                mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
```

5. Bonus Items

5a. `subset()`

`subset()` is a specialised shorthand function for subsetting data frames, and saves some typing because you don't need to repeat the name of the data frame.

```
subset(mtcars, gear == 5)
```

```
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
```

```
subset(mtcars, gear == 5 & cyl == 4)
```

```
##                mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
```

5b. Simplifying vs. preserving subsetting

It's important to understand the distinction between simplifying and preserving subsetting. Simplifying subsets returns the simplest possible data structure that can represent the output, and is useful interactively because it usually gives you what you want. Preserving subsetting keeps the structure of the output the same as the input, and is generally better for programming because the result will always be the same type. Omitting `drop = FALSE` when subsetting matrices and data frames is one of the most common sources of programming errors. (It will work for your test cases, but then someone will pass in a single column data frame and it will fail in an unexpected and unclear way.)

Unfortunately, how you switch between simplifying and preserving differs for different data types, as summarised in the table below.

	Simplifying	Preserving
Vector	`x[[1]]`	`x[1]`
List	`x[[1]]`	`x[1]`
Factor	`x[1:4, drop = T]`	`x[1:4]`
Data frame	`x[, 1]` **or** `x[[1]]`	`x[, 1, drop = F]` **or** `x[1]`

Preserving is the same for all data types: you get the same type of output as input. Simplifying behaviour varies slightly between different data types, as described below:

- **Atomic vector**: removes names.

```
x <- c(a = 1, b = 2)
x
```

```
## a b
## 1 2
```

```
x[1]
```

```
## a
## 1
```

```
x[[1]]
```

```
## [1] 1
```

- **List**: return the object inside the list, not a single element list.

```
y <- list(a = 1, b = 2)
y
```

```
## $a
## [1] 1
##
## $b
## [1] 2
```

```
str(y[1])
```

```
## List of 1
##  $ a: num 1
```

```
str(y[[1]])
```

```
##  num 1
```

- **Factor**: drops any unused levels.

```
z <- factor(c("a", "b"))
z[1]
```

```
## [1] a
## Levels: a b
```

```
z[1, drop = TRUE]
```

```
## [1] a
## Levels: a
```

- **Matrix**: if any of the dimensions has length 1, drops that dimension.

  ```
  a <- matrix(1:4, nrow = 2)
  a
  ```

  ```
  ##      [,1] [,2]
  ## [1,]    1    3
  ## [2,]    2    4
  ```

  ```
  a[1, , drop = FALSE]
  ```

  ```
  ##      [,1] [,2]
  ## [1,]    1    3
  ```

  ```
  a[1, ]
  ```

  ```
  ## [1] 1 3
  ```

- **Data frame**: if output is a single column, returns a vector instead of a data frame.

  ```
  df <- data.frame(a = 1:2, b = 3:4)
  df
  ```

  ```
  ##   a b
  ## 1 1 3
  ## 2 2 4
  ```

```r
str(df[1])
```

```
## 'data.frame':    2 obs. of  1 variable:
##  $ a: int  1 2
```

```r
str(df[[1]])
```

```
##  int [1:2] 1 2
```

```r
str(df[, "a", drop = FALSE])
```

```
## 'data.frame':    2 obs. of  1 variable:
##  $ a: int  1 2
```

```r
str(df[, "a"])
```

```
##  int [1:2] 1 2
```