Subsetting

Stats 102A

Miles Chen

Acknowledgements: Michael Tsiang and Hadley Wickham

Week 1 Friday

UCLA

Section 1

Subsetting Atomic Vectors

Subsetting 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)
```

We start with a simple vector `x`, that has been crafted so that the number after the decimal point gives the original position in the vector.

There are a few ways you to subset a vector:

- Positive Integers
- Negative Integers
- Logical Vectors
- Character Vectors

Subsetting with Positive Integers

Positive integers return elements at the specified positions:

```
# x <- c(2.1, 4.2, 3.3, 5.4)
x[c(3, 1)]
```

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

```
order(x)
```

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

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

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

Subsetting with Positive Integers

```
# x <- c(2.1, 4.2, 3.3, 5.4)
# Duplicated indices yield duplicated values
x[c(1, 1)]
```

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

```
# Real numbers are silently truncated to integers
x[c(2.1, 2.9)]
```

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

Subsetting with Negative Integers

Negative integers omit elements at the specified positions:

```
# x <- c(2.1, 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(-1, 2)]
```

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

Subsetting with Logical vectors

Logical vectors select elements where the corresponding logical value is `TRUE`. 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)
x[c(TRUE, TRUE, FALSE, FALSE)]
```

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

```
x[x > 3]
```

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

Subsetting with Logical vectors

If the logical vector is shorter than the vector being subsetted, it will be *recycled* to be the same length.

```
# x <- c(2.1, 4.2, 3.3, 5.4)
x[c(TRUE, FALSE)]
```

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

```
# The above is equivalent to x[c(TRUE, FALSE, TRUE, FALSE)]
```

A missing value in the index always yields a missing value in the output:

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

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

```
x[c(TRUE, NA)] # recycling rules still apply
```

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

Special Cases

Nothing returns the original vector. This is not useful for vectors but is very useful for matrices, data frames, and arrays. It can also be useful in conjunction with assignment.

```
x[]
```

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

Zero returns a zero-length vector. This is not something you usually do on purpose, but it can be helpful for generating test data.

```
x[0]
```

```
## numeric(0)
```

Subsetting with Character vectors

If the vector is named, you can also use **Character vectors** to return elements with matching names.

```
(y <- setNames(x, letters[1:4]))
```

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

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

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

Subsetting with Character vectors

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

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

```
# When subsetting with [ names must be spelled exactly to find a match
z <- c(abc = 1, def = 2)
z[c("a", "d")]
```

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

Useful application: Lookup tables (character subsetting)

Character matching provides a powerful way to make lookup tables.

```
x <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "Male", f = "Female", u = NA)
lookup[x]  # subset the labeled vector with the vector of abbreviations
```

```
##        m        f        u        f        f        m        m
##   "Male" "Female"       NA "Female" "Female"   "Male"   "Male"
```

```
# we can clean up the resulting vector by removing names
unname(lookup[x])
```

```
## [1] "Male"   "Female" NA       "Female" "Female" "Male"   "Male"
```

Section 2

Subsetting Lists

Subsetting Lists

Subsetting a list works in the same way as subsetting an atomic vector.

Important: Using a single square bracket `[` will always return a list.

Using a double square bracket `[[` or dollar-sign `$`, as described next, let you pull out the components of the list.

Subsetting operators

There subsetting operators: `[[ ]]` and `$` are similar to `[ ]`, except it can only return a single object and it allows you to pull pieces out of a list. `$` 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

Double square brackets

Because it can return only a single object, you must use [[ ]] with either a single positive integer or a single string

```
a <- list(x = 1:4, y = 9:6)
a
```

```
## $x
## [1] 1 2 3 4
##
## $y
## [1] 9 8 7 6
```

```
a[[1]]
```

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

```
a[["y"]]
```

```
## [1] 9 8 7 6
```

Single vs double square brackets

```
a[c("y", "x", "x")] # can use single brackets with a vector of multiple items
```

```
## $y
## [1] 9 8 7 6
##
## $x
## [1] 1 2 3 4
##
## $x
## [1] 1 2 3 4
```

```
a[[ c("y", "x") ]] # this does not work
```

```
## Error in a[[c("y", "x")]]: subscript out of bounds
```

Sidenote: Recursive Subsetting

Putting a vector of multiple elements in double square brackets performs recursive subsetting. `a[[c("y","x")]]` is actually equivalent to `a[["y"]][["x"]]` which only works if `a` is a list with element `y`, and `y` itself has an element inside it called `x`.

```
a <- list(x = 1:4, y = list(x = "a", z = "b"))
str(a)
```

```
## List of 2
##  $ x: int [1:4] 1 2 3 4
##  $ y:List of 2
##   ..$ x: chr "a"
##   ..$ z: chr "b"
```

```
a[["y"]][["x"]]
```

```
## [1] "a"
```

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

```
## [1] "a"
```

Another Example

```
d <- list(
  a = c(1, 2, 3),
  b = c(TRUE, TRUE, FALSE),
  c = c("a")
  )
str(d)
```

```
## List of 3
##  $ a: num [1:3] 1 2 3
##  $ b: logi [1:3] TRUE TRUE FALSE
##  $ c: chr "a"
```

Single square bracket vs double square bracket

```
d[1] # single square bracket returns a list containing the first element
```

```
## $a
## [1] 1 2 3
```

```
typeof(d[1])
```

```
## [1] "list"
```

```
d[[1]] # double square bracket returns the contents of the first element
```

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

```
typeof(d[[1]])
```

```
## [1] "double"
```

Let's make a new list

```r
l1 <- list(
  1:8,
  letters[1:4],
  5:1
  )  # The list has three elements, but they are unnamed
str(l1)
```

```
## List of 3
##  $ : int [1:8] 1 2 3 4 5 6 7 8
##  $ : chr [1:4] "a" "b" "c" "d"
##  $ : int [1:5] 5 4 3 2 1
```

Single square bracket vs Double square bracket

```
l1[1] # this is a list. returns the first train car
```

```
## [[1]]
## [1] 1 2 3 4 5 6 7 8
```

```
l1[[1]] # this is a vector. the contents of the first train car
```

```
## [1] 1 2 3 4 5 6 7 8
```

List subsetting

```
l1[1][2]    # l1[1] is a list of one item. It has no second element
```

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

```
l1[[1]][2] # l1[[1]] is the integer vector 1:8. The second element is 2.
```

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

```
l1[[c(1,2)]] # Recursive subsetting: l1[[c(1,2)]] is equal to l1[[1]][[2]]
```

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

A list inside a list (A train inside a train car)

```
# l2 has 3 elements: the first is the list l1
# the second and third elements are vectors
l2 <- list( l1, c(10, 20, 30), LETTERS[4:9])
str(l2)
```

```
## List of 3
##  $ :List of 3
##   ..$ : int [1:8] 1 2 3 4 5 6 7 8
##   ..$ : chr [1:4] "a" "b" "c" "d"
##   ..$ : int [1:5] 5 4 3 2 1
##  $ : num [1:3] 10 20 30
##  $ : chr [1:6] "D" "E" "F" "G" ...
```

Single square bracket returns the list (train car with the train inside)

```
# subsetting with a single square bracket returns a list of one element
# which itself contains the list l1
l2[1]
```

```
## [[1]]
## [[1]][[1]]
## [1] 1 2 3 4 5 6 7 8
##
## [[1]][[2]]
## [1] "a" "b" "c" "d"
##
## [[1]][[3]]
## [1] 5 4 3 2 1
```

Single square bracket returns the list (train car with the train inside)

```
# str shows that it is a list in a list
str(l2[1])
```

```
## List of 1
##  $ :List of 3
##   ..$ : int [1:8] 1 2 3 4 5 6 7 8
##   ..$ : chr [1:4] "a" "b" "c" "d"
##   ..$ : int [1:5] 5 4 3 2 1
```

Double square bracket returns the contents (the actual train inside)

```
# double square bracket returns the contents of the first element
# which is the list l1 (just the listt)
l2[[1]]
```

```
## [[1]]
## [1] 1 2 3 4 5 6 7 8
##
## [[2]]
## [1] "a" "b" "c" "d"
##
## [[3]]
## [1] 5 4 3 2 1
```

Double square bracket returns the contents (the actual train inside)

```
# str reveals we have just the list, not a list in a list
str(l2[[1]])
```

```
## List of 3
##  $ : int [1:8] 1 2 3 4 5 6 7 8
##  $ : chr [1:4] "a" "b" "c" "d"
##  $ : int [1:5] 5 4 3 2 1
```

Pay attention to the differences

```
l2[[1]][1]
```

```
## [[1]]
## [1] 1 2 3 4 5 6 7 8
```

```
l2[[1]][1][2]
```

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

```
l2[[1]][[1]]
```

```
## [1] 1 2 3 4 5 6 7 8
```

```
l2[[1]][[1]][2]
```

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

Double square brackets on atomic vectors

Double square brackets with atomic vectors behave similarly to the use of single square brackets with a few key differences, particularly in handling out-of-bounds indexes.

```
x <- c("a" = 1, "b" = 2, "c" = 3)
x[1] # single square brackets preserve names
```

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

```
x[[1]] # double square brackets drop names
```

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

Double square brackets on atomic vectors

```
x <- 1:3
x[5] # single square brackets return NA for out-of-bounds index
```

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

```
x[[5]] # double square brackets return error for out-of-bounds index
```

```
## Error in x[[5]]: subscript out of bounds
```

```
x[NULL] # single square brackets return length 0 vectors for 0 or NULL
```

```
## integer(0)
```

```
x[[NULL]] # double square brackets return error for 0 or NULL
```

```
## Error in x[[NULL]]: attempt to select less than one element in get1index
```

Single vs Double brackets for OOB and NULL indices

Similar rules for out-of-bounds indexes happen with single vs double square brackets with lists. Let's take a look at l1 again.

```
l1
```

```
## [[1]]
## [1] 1 2 3 4 5 6 7 8
##
## [[2]]
## [1] "a" "b" "c" "d"
##
## [[3]]
## [1] 5 4 3 2 1
```

Single vs Double brackets for OOB and NULL indices

```
l1[4] # out of bounds returns a train car with NULL inside
```

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

```
l1[[4]] # common error in for loops
```

```
## Error in l1[[4]]: subscript out of bounds
```

```
l1[NULL] # NULL or 0 returns no train cars
```

```
## list()
```

```
l1[[NULL]]
```

```
## Error in l1[[NULL]]: attempt to select less than one element in get1index
```

Section 3

Subsetting Matrices and arrays

Subsetting Matrices and arrays

You can subset higher-dimensional structures in three ways:

- With multiple vectors. (Most common method)
- With a single vector.
- With a matrix. (least common)

The most common way of subsetting matrices (2d) and arrays (>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.

Let's create a matrix

```
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
```

Subsetting the matrix

```
a[1:2, ] # first and second rows
```

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

```
a[c(T, F, T), c("B", "A")] # first and third rows, columns b and a only
```

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

```
a[0, -2] # no rows, all but the second column
```

```
##      A C
```

Subsetting a matrix simplifies by default

By default, single square bracket subsetting [will simplify the results to the lowest possible dimensionality. We will later discuss preservation to avoid this.

```
is.vector(a) # matrix is not a vector
```

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

```
a[1,] # no longer a matrix
```

```
## A B C
## 1 4 7
```

```
is.vector(a[1,]) # when you subset the row, it becomes a vector
```

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

Matrices are atomic vectors

Because matrices and arrays are implemented as vectors with special attributes, you can subset them with a single vector. In that case, they will behave like a vector. Arrays in R are stored in column-major order:

```
(vals <- matrix(LETTERS[1:25], nrow = 5))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "A"  "F"  "K"  "P"  "U"
## [2,] "B"  "G"  "L"  "Q"  "V"
## [3,] "C"  "H"  "M"  "R"  "W"
## [4,] "D"  "I"  "N"  "S"  "X"
## [5,] "E"  "J"  "O"  "T"  "Y"
```

```
# select the 8th and 9th values in the vector
vals[c(8, 9)]
```

```
## [1] "H" "I"
```

Subset a matrix with a matrix

You can also subset higher-dimensional data structures with an integer matrix (or, if named, a character matrix). Each row in the matrix specifies the location of one value, where each column corresponds to a dimension in the array being subsetted. This means that you use a 2 column matrix to subset a matrix, a 3 column matrix to subset a 3d array, and so on. The result is a vector of values:

```
select <- matrix(ncol = 2, byrow = TRUE, c(
  1, 5,   ## select the values at the coordinates (1,5)
  3, 1,   ## value at coord (3, 1), third row, 1st col
  2, 3,
  1, 1
))
vals[select]
```

```
## [1] "U" "C" "L" "A"
```

Subset a 3d array with a matrix with 3 columns

```r
# generalizes to three dimensions
(ar <- array(1:12, c(2,3,2)))
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

Subset a 3d array with a matrix with 3 columns

```
select_ar <- matrix(ncol = 3, byrow = TRUE, c(
  1,2,1,
  2,3,2))
ar[select_ar]
```

```
## [1]  3 12
```

Section 4

Subsetting Data Frames

Data frames

Data frames possess the characteristics of both lists and matrices: if you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices.

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

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

```
df[df$y %% 2 == 0, ]  # choose the rows where this logical statement is TRUE
```

```
##   x y z
## 1 1 4 a
## 3 3 2 c
```

Subsetting Data Frames like a list

```
# Select columns like you would a list:
df[c("x", "z")]
```

```
##   x z
## 1 1 a
## 2 2 b
## 3 3 c
## 4 4 d
```

Subsetting Data Frames like a matrix

```
# select columns like a matrix
df[ , c("x", "z")]
```

```
##   x z
## 1 1 a
## 2 2 b
## 3 3 c
## 4 4 d
```

Subsetting Data Frames to select rows

To select the rows, you can provide a vector before the comma.

Here we choose the first and third rows.

```
df[c(1, 3), ]  # note the comma
```

```
##   x y z
## 1 1 4 a
## 3 3 2 c
```

Subsetting Data Frames to select rows

If you leave out the comma, it will try to subset the data frame like a list.

In this case, we get the first and third columns and all the rows.

```
df[c(1, 3)]  # comma is missing
```

```
##   x z
## 1 1 a
## 2 2 b
## 3 3 c
## 4 4 d
```

Data Frames with named columns

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

```
str(df["x"])  # preserves: remains a data frame
```

```
## 'data.frame':    4 obs. of  1 variable:
##  $ x: int  1 2 3 4
```

```
str(df[ , "x"])  # simplifies: becomes a vector
```

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

Data Frames with named columns

```
str(df$x)  # dollar sign always simplifies: becomes a vector
```

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

```
str(df[["x"]]) # simplifies: becomes a vector
```

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

Simplifying vs. preserving

When you subset, R often simplifies the result to an atomic vector or a form different from the original form. We saw this with the data frames in previous slides.

The next few slides cover simplifying vs preserving behaviors in R for different data types

Simplifying vs. preserving: Atomic vectors

For atomic vectors simplifying removes names

```
x <- c(a = 1, b = 2)
x[1]  # preserving: keeps names
```

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

```
x[[1]] # simplifying: drops names
```

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

Simplifying vs. preserving: Lists

Simplifying return the object inside the list, not a single element list.

```
y <- list(a = 1, b = 2)
str(y[1])  # preserving: still a list
```

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

```
str(y[[1]]) # simplifying: is a vector
```

```
##  num 1
```

Simplifying vs. preserving: Factors

Factors are a special case. For factors, simplifying is achieved by using the argument `drop = TRUE` inside the square brackets. It drops any unused levels.

```
z <- factor(c("a", "b"))
z[1] # preserving: keeps levels that do not appear
```

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

```
z[1, drop = TRUE] # simplifying: drops levels that no longer appear
```

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

Simplifying vs. preserving: Matrices and arrays

If subsetting a matrix or array results in a dimension with a length 1, it will drop that dimension. For example, when you subset a row from a matrix, R will return an atomic vector rather than a 1 x n matrix. If you want to preserve the matrix structure, use the `drop = FALSE` argument inside the square brackets.

```
a <- matrix(1:4, nrow = 2)
a[1, , drop = FALSE] # preserving: keeps matrix structure
```

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

```
a[1, ] # simplifying: returns an atomic vector
```

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

Simplifying vs. preserving: Data Frames

Data Frames are lists, so when you subset one like a list, the simplifying vs preserving rules apply.

```
df <- data.frame(a = 1:2, b = 1:2)
str(df[1]) # preserving: a single square bracket returns a data frame
```

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

```
str(df[[1]]) # simplifying : double brackets returns a vector
```

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

Examples of subsetting

```
head(mtcars, 10)
```

```
##                    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
```

What's wrong?

```
mtcars[mtcars$cyl <= 5]
```

```
## Error in `[.data.frame`(mtcars, mtcars$cyl <= 5): undefined columns selecte
```

The fix

```
# need to specify selection of rows with a comma
mtcars[mtcars$cyl <= 5, ]
```

```
##                  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Datsun 710      22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## 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
## 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
## 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
## Volvo 142E      21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

What's wrong?

```
mtcars[mtcars$cyl == 4 | 6, ]
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4
## Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4
## Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3
## Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3
```

The fix

```r
# the mtcars$cyl == 4 | 6
# on the left is a logical vector
# on the right of 'or |' is the number 6 which gets coerced to TRUE,
# so it returns TRUE for everything

# the or operator has to be between two logical vectors
mtcars[mtcars$cyl == 4 | mtcars$cyl == 6, ]
```

```
##                   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
## Valiant          18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## 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
```

What's wrong?

```
mtcars[1:13]
```

```
## Error in `[.data.frame`(mtcars, 1:13): undefined columns selected
```

Don't forget the comma!

```
mtcars[1:13,] # without the comma, it tried to select the first 13 names
```

```
##                    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
```