Advanced R by Hadley Wickham

Chapter 4: Subsetting

Scott Nestler

@ScottNestler

2020-04-23

What's in Chapter 4

- Section 4.1: Introduction & Key Points
- Section 4.2: Selecting multiple elements
- Section 4.3: Selecting a single element
- Section 4.4: Subsetting and assignment
- Section 4.5: Applications (Using subsetting to solve problems)

Introduction & Key Points

- There are 6 ways to subset atomic vectors (more in 4.2 & 4.3)
- There are 3 subsetting operators: [[, [, and `$`
- Subsetting operators interact differently with various vector types (e.g. atomic vectors, lists, factors, matrices, and data frames)
- Subsetting and assignment can be combined ("subsassignment")
- Subsetting complements structure, or `str()`, which shows you *all* the pieces of an object, but subsetting lets you pull out only the pieces you are interested in
- Often useful to use RStudio Viewer, with `View(my_object)` to know which pieces you want to subset

Selecting Multiple Elements

- Use `[` to select any number of elements from a vector (of any type), in one of 6 ways
- We will use the following vector (from the beer dataset) in the examples

```
library(dplyr)
brewer_size <- readr::read_csv('https://raw.githubusercontent.com/rfordatascience
brewers <- brewer_size %>%
  group_by(year) %>%
  summarize(sum = sum(n_of_brewers), barrels = sum(total_barrels)) %>%
  select(sum, barrels) %>%
  c()
n_brewers <- brewers$sum
n_barrels <- brewers$barrels
n_brewers
```

```
##  [1]  3556  3628  4186  4860  5692  6746  9008 10192 11296 11928 12800
```

```
n_barrels
```

```
##  [1] 393938550 390422806 385493644 392294405 383994749 384030276 382226036
##  [8] 379679827 371163674 365581921        NA
```

Selecting Multiple Elements: Approaches 1 & 2

- **Positive integers** return elements at the specified position(s).

```
n_brewers[2] # Can also be used for a single element
```

```
## [1] 3628
```

```
n_brewers[c(1,11,12)] # Note what happens with out of bounds index
```

```
## [1]  3556 12800    NA
```

```
n_brewers[4:6]
```

```
## [1] 4860 5692 6746
```

- **Negative integers** *exclude* elements at the specified position(s).

```
n_brewers[-11] # Omits the last (11th) value
```

```
##  [1]  3556  3628  4186  4860  5692  6746  9008 10192 11296 11928
```

```
n_brewers[c(-1:-5)] # Omits the first 5 values
```

```
## [1]  6746  9008 10192 11296 11928 12800
```

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

Selecting Multiple Elements: Approaches 3 & 4

- **Logical vectors** select elements where the corresponding logical value is true. Perhaps the most useful type of subsetting.

```
n_brewers[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE
```

```
## [1]  3556  4186  5692  9008 11296 12800
```

```
n_brewers[n_brewers > 10000]
```

```
## [1] 10192 11296 11928 12800
```

```
n_brewers[n_brewers > 4000 & n_brewers < 9000]
```

```
## [1] 4186 4860 5692 6746
```

```
n_brewers[c(TRUE, TRUE, FALSE)] # Usual recycling rules apply
```

```
## [1]  3556  3628  4860  5692  9008 10192 11928 12800
```

- **Nothing** returns the original vector. Useful for matricies, data frames, and arrays (later).

```
n_brewers[]
```

```
##  [1]  3556  3628  4186  4860  5692  6746  9008 10192 11296 11928 12800
```

- More useful with matrices, data frames, and arrays.

Selecting Multiple Elements: Approaches 5 & 6

- **Zero** return returns a zero length vector. Useful for generating test data.

```
n_brewers[0]
```

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

- Helpful for generating test data
- **Character vectors** return elements of vectors *with names*.

```
nb <- setNames(n_brewers, letters[1:11])
nb[c("a", "b", "c")]
```

```
##    a    b    c
## 3556 3628 4186
```

```
nb[c("b", "b")] # Can repeat indices
```

```
##    b    b
## 3628 3628
```

- Recommendation is to not subset with factors

Selecting Multiple Elements of Lists, Matrices, and Arrays

- For Lists, it's the same as for an atopmic vector
  - `[` always returns a list; `[[` and `$` let you pull out elements (later)
- For **matrices** and **arrays**, there are 3 ways:
  - With multiple vectors
  - With a single vector
  - With a matrix

```
A <- matrix(n_brewers[1:10], nrow = 2)
A
```

```
##      [,1] [,2] [,3]  [,4]  [,5]
## [1,] 3556 4186 5692  9008 11296
## [2,] 3628 4860 6746 10192 11928
```

```
A[1,] # To select first row
```

```
## [1]  3556  4186  5692  9008 11296
```

```
A[,2:3] # To select 2nd and 3rd columns
```

```
##      [,1] [,2]
## [1,] 4186 5692
## [2,] 4860 6746
```

```
A[2,4] # Can also select a single element
```

```
## [1] 10192
```

Selecting Multiple Elements of Data Frames

- Data frames have characteristics of both lists and matrices, so:
  - When subsetting with a single index, they behave like lists.
  - When subsetting with two indices, they behave like matrices.

```
B <- data.frame(year = 2009:2013, num = n_brewers[1:5], prod = n_barrels[1:5])
B[1:2] # Selects first two columns
```

```
##   year  num
## 1 2009 3556
## 2 2010 3628
## 3 2011 4186
## 4 2012 4860
## 5 2013 5692
```

```
B[2,] # Selects the second row
```

```
##   year  num      prod
## 2 2010 3628 390422806
```

```
B[4,2] # Select a single element
```

```
## [1] 4860
```

```
str(B["year"]) # list subsetting does not simplify
```

```
## 'data.frame':    5 obs. of  1 variable:
##  $ year: int  2009 2010 2011 2012 2013
```

```
str(B[, "year"]) # matrix subsetting simplifies by default
```

```
##  int [1:5] 2009 2010 2011 2012 2013
```

Selecting Multiple Elements of Tibbles

- Subsetting a tibble with `[` always returns a tibble.

```
C <- tibble::tibble(B)
C[2,] # Selects the second row
C[4,2] # Select a single element
str(C["year"]) # Same as next
str(C[,"year"]) # Same as previous
```

```
## # A tibble: 1 x 3
##    year   num       prod
##   <int> <dbl>      <dbl>
## 1  2010  3628 390422806.
## # A tibble: 1 x 1
##     num
##   <dbl>
## 1  4860
## tibble [5 x 1] (S3: tbl_df/tbl/data.frame)
##  $ year: int [1:5] 2009 2010 2011 2012 2013
## tibble [5 x 1] (S3: tbl_df/tbl/data.frame)
##  $ year: int [1:5] 2009 2010 2011 2012 2013
```

Preserving Dimensionality

- Subsetting a matrix or data frame with a single number will (by default) simplify the returned output to an object with lower dimensionality
- You can change this with `drop = FALSE`.
- Tibbles default to `drop = FALSE` because the default `drop = TRUE` behavior is a common source of bugs in functions.
- Factor subsetting is different; it affects levels (not dimensions) and defaults to `FALSE`.
- If you are using `drop = TRUE` a lot, you shoudl probably use a character vector instead of a factor.

```
str(A[1,])
```

```
##  num [1:5] 3556 4186 5692 9008 11296
```

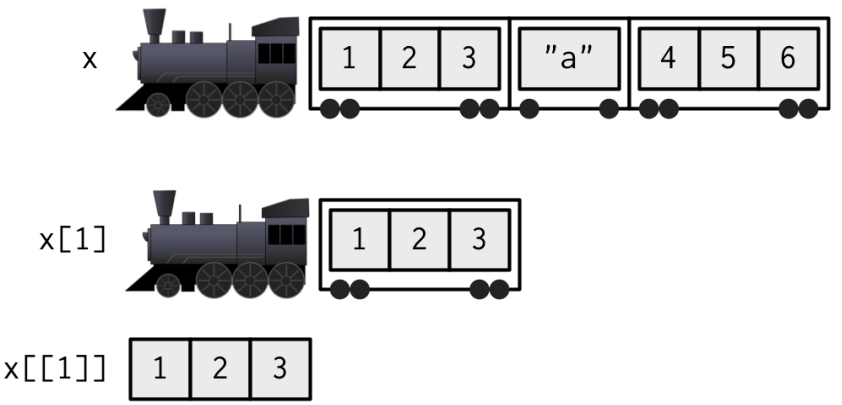
```
str(A[1, , drop = FALSE])
```

```
##  num [1, 1:5] 3556 4186 5692 9008 11296
```

Selecting a single element

- Two subsetting operators
  - `[[` is used for extracing single elements
  - `$` is just shorthand operator, i.e. x$y is the same as x[["y"]] (in 5 fewer keystrokes)
  - Primary use case for `[[` is when working with lists, as you get a list back.
  - The "train of cars" example is useful in illustrating the difference between `[` and `[[`.

```
x <- list(1:3, "a", 4:6)
```



More on [[

- `[[` can only return a single item, so you can only use it with a single positive integer or a single string.
- If you use a vector with `[[`, it will subset recursively.
    - This means that `x[[c(1,2)]]` is equivalent to `x[[1]][[2]]`
- You have to use `[[` when workign with lists, but it is recommended for extracting a single value from atomic vectors too.
- For somethign like this, it probably doesn't matter.

```
n_brewers[7]
```

```
## [1] 9008
```

```
n_brewers[[7]]
```

```
## [1] 9008
```

- But in code like this, somehow it does?

```
for (i in 2:length(x)) {
  out[i] <- fun(x[i], out[i-1])
}

for (i in 2:length(x)) {
  out[[i]] <- fun(x[[i]], out[[i-1]])
}
```

Now about $

- As mentioned, it is a shortcut, with `x$y` being *roughly equivalent* to x[["y"]].
- Often used to access variables in data frames.
- A common mistake is using it when the name of a column is stored in a variable.
- An important difference betweeen `$` and `[[` is that `$` does partial (left to right) matching.
- A common mistake with `$` is using it when you have the name of a column stored in a variable

```
# Tried to put an example here with the beer data set but ran out of time
# and refused to use the mtcars example from the book.  Will add in later.
```

Missing and Out-of-Bounds Indices

- What happens when you use an "invalid" index with `[[`?
- Inconsistencies in this table led to the development of `purr::pluck()` and `purr::chuck()`.

Table from 4.3.3

row[[col]]	Zero-length	OOB(int)	OOB(chr)	Missing
Atomic	Error	Error	NULL	NULL
Error	Error	Error	NULL	NULL
Error	List	NULL	NULL	NULL

@ and `slot()` Operators

- These operators are needed for S4 objects (covered in Chap. 15)
- @ is the equivalent of $; @ is more restrictive
- `slot()` is the equivalent to `[[`

Subsetting and Assignment ("Subassignment")

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

```
n_brewers
```

```
##  [1]  3556  3628  4186  4860  5692  6746  9008 10192 11296 11928 12800
```

```
n_brewers[5] <- 42
n_brewers[c(1,2)] <- c(1.618, 186282)
n_brewers
```

```
##  [1]      1.618 186282.000   4186.000   4860.000     42.000   6746.000
##  [7]   9008.000  10192.000  11296.000  11928.000  12800.000
```

- Be sure that the length(value) is the same length of x[i] due to complex recycling rules.

Application - Lookup Tables

Character Subsetting

- Character matching can be used to create lookup tables.
- You can use `unname()` to remove the names if you want

```
brewing_materials <- readr::read_csv('https://raw.githubusercontent.com/rfordatas
types <- substring(brewing_materials$type[1:15],1,1)
types
```

```
##  [1] "M" "C" "R" "B" "W" "T" "S" "H" "H" "O" "T" "T" "M" "C" "R"
```

```
lookup <- c(M = "Malt", C = "Corn", R = "Rice", B = "Barley", W = "Wheat", T = "T
lookup[types]
```

```
##        M        C        R        B        W        T        S        H
##   "Malt"   "Corn"   "Rice" "Barley"  "Wheat"  "Total"  "Sugar"   "Hops"
##        H        O        T        T        M        C        R
##   "Hops"  "Other"  "Total"  "Total"   "Malt"   "Corn"   "Rice"
```

```
unname(lookup[types])
```

```
##  [1] "Malt"   "Corn"   "Rice"   "Barley" "Wheat"  "Total"  "Sugar"  "Hops"
##  [9] "Hops"   "Other"  "Total"  "Total"  "Malt"   "Corn"   "Rice"
```

Application - Matching and Merging By Hand

Integer Subsetting

- Can have more complicated lookup tables with multiple columns.
- You can use `unname()` to remove the names if you want

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(
  grade = 3:1,
  desc = c("Excellent", "Good", "Poor"),
fail = c(F, F, T)
)
id <- match(grades, info$grade)
id
```

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

```
info[id, ]
```

```
##     grade      desc  fail
## 3       1      Poor  TRUE
## 2       2      Good FALSE
## 2.1     2      Good FALSE
## 1       3 Excellent FALSE
## 3.1     1      Poor  TRUE
```

Application - Random Samples and Bootstraps

IntegerSubsetting

- Can use integer indices to randomly sample or bootstrap a vector or data frame using `sample()`.

```
B <- data.frame(year = 2009:2013, num = n_brewers[1:5], prod = n_barrels[1:5])
B
```

```
##   year        num      prod
## 1 2009      1.618 393938550
## 2 2010 186282.000 390422806
## 3 2011   4186.000 385493644
## 4 2012   4860.000 392294405
## 5 2013     42.000 383994749
```

```
# Randomly reorder
B[sample(nrow(B)), ]
```

```
##   year        num      prod
## 4 2012   4860.000 392294405
## 3 2011   4186.000 385493644
## 1 2009      1.618 393938550
## 2 2010 186282.000 390422806
## 5 2013     42.000 383994749
```

Application - Random Samples and Bootstraps (cont.)

Integer Subsetting

- Can use integer indices to randomly sampel or bootstrap a vector or data frame using `sample()`.

```
# Select 3 random rows
B[sample(nrow(B), 2), ]
```

```
##   year  num      prod
## 5 2013   42 383994749
## 3 2011 4186 385493644
```

```
#Select 10 bootstrap replicates
B[sample(nrow(B), 10, replace = TRUE), ]
```

```
##     year        num      prod
## 2   2010 186282.000 390422806
## 4   2012   4860.000 392294405
## 2.1 2010 186282.000 390422806
## 2.2 2010 186282.000 390422806
## 3   2011   4186.000 385493644
## 4.1 2012   4860.000 392294405
## 3.1 2011   4186.000 385493644
## 2.3 2010 186282.000 390422806
## 5   2013     42.000 383994749
## 1   2009      1.618 393938550
```

```
# The number after decimal indicates additional occurrences
```

Application - Ordering

Integer Subsetting

- Provide a vector to `order()` and it returns an integer vector describing how to order the subsetted vector.
- To break ties, you can supply additional variables to `order()`.

```
types
```

```
##  [1] "M" "C" "R" "B" "W" "T" "S" "H" "H" "O" "T" "T" "M" "C" "R"
```

```
order(types)
```

```
##  [1]  4  2 14  8  9  1 13 10  3 15  7  6 11 12  5
```

```
types[order(types)]
```

```
##  [1] "B" "C" "C" "H" "H" "M" "M" "O" "R" "R" "S" "T" "T" "T" "W"
```

```
types[order(types, decreasing = TRUE)] # Change order from ascending to descendi
```

```
##  [1] "W" "T" "T" "T" "S" "R" "R" "O" "M" "M" "H" "H" "C" "C" "B"
```

Application - Ordering (cont.)

Integer Subsetting

- For two or more dimensions, `order()` makes it easy to order *either* the rows or columns of an object.

```
# Reorder by 'num' column
B[order(B$num), ]
```

```
##   year        num      prod
## 1 2009      1.618 393938550
## 5 2013     42.000 383994749
## 3 2011   4186.000 385493644
## 4 2012   4860.000 392294405
## 2 2010 186282.000 390422806
```

```
# Reorder by names of columns
B[, order(names(B))]
```

```
##          num      prod year
## 1      1.618 393938550 2009
## 2 186282.000 390422806 2010
## 3   4186.000 385493644 2011
## 4   4860.000 392294405 2012
## 5     42.000 383994749 2013
```

Application - Expanding Aggregated Counts

Character Subsetting

- Given a data frame where identical rows have been collapsed into one and a count column, use `rep()` and integer subsetting to uncollapse.
- This works because `rep(x, y)` repeats `x[i] y[i]` times.

```
df <- data.frame(x = c(2, 4, 1), y = c(9, 11, 6), n = c(3,5,1))
rep(1:nrow(df), df$n)
```

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

```
df[rep(1:nrow(df), df$n), ]
```

```
##     x  y n
## 1   2  9 3
## 1.1 2  9 3
## 1.2 2  9 3
## 2   4 11 5
## 2.1 4 11 5
## 2.2 4 11 5
## 2.3 4 11 5
## 2.4 4 11 5
## 3   1  6 1
```

Application - Removing Columns From Data Frames

Character Subsetting

- There are two ways to remove columns from a data frame.
    - You can set individual columns to `NULL`.
    - Or you can subset to return only the columns you want.
    - If you only know the columns you **don't** want, use set operations to work out which columns to keep:

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

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

```
##   x y
## 1 1 3
## 2 2 2
## 3 3 1
```

```
df[setdiff(names(df), "z")]
```

```
##   x y
## 1 1 3
## 2 2 2
## 3 3 1
```

Application - Selecting Rows Based on a Condition

Logical Subsetting

- To combine conditions from multiple columns use logical subsetting

```
B <- data.frame(year = 2009:2013, num = n_brewers[1:5], prod = n_barrels[1:5])
B[B$num >= 4000,]
```

```
##   year    num      prod
## 2 2010 186282 390422806
## 3 2011   4186 385493644
## 4 2012   4860 392294405
```

```
B[B$num >= 4000 & B$year < 2013,]
```

```
##   year    num      prod
## 2 2010 186282 390422806
## 3 2011   4186 385493644
## 4 2012   4860 392294405
```

Application - Boolean Algebra vs Sets

Logical & Integer Subsetting

- Using set operations is useful when:
  - You want to find the first (or last) `TRUE`.
  - You have very few `TRUE`s and very few `FALSE`s, so a set representation is faster and uses less storage
- Use `which()` to convert a Boolean representation to an integer representation.
- There is no reverse operation in base R, but it can be written like this:

```
x <- sample(10) < 4
which(x)
```

```
## [1] 2 6 8
```

```
unwhich <- function(x, n) {
  out <- rep_len(FALSE, n)
  out[x] <- TRUE
  out
}

unwhich(which(x), 10)
```

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

Application - Boolean Algebra vs Sets (cont.)

Logical & Integer Subsetting

- Here are two logical vectors and some operations useful in subsetting.

```
x1 <- 1:10 %% 2 == 0 # uses the mod operator `%%` to identify even numbers
x2 <- which(x1)
y1 <- 1:10 %% 5 == 0 # Finds numbers divisible by 5 with remainder 0
y2 <- which(y1)

x1 & y1 # gives the intersection
```

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

```
intersect (x2, y2) # check it this way; gives even numbers
```

```
## [1] 10
```

```
x1 | y1 # gives the union
```

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

```
union(x2, y2) # provides even numbers and those divisible by 5
```

```
## [1]  2  4  6  8 10  5
```

```
x1 & !y1 # gives even numbers *not* divisible by 5
```

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

```
setdiff(x2, y2) # check it this way
```

```
## [1] 2 4 6 8
```

Application - Boolean Algebra vs Sets (cont. cont.)

Logical & Integer Subsetting

- Common mistake is using `x[which(y)]` instead of `x[y]`. Problem is that the `which()` switches from logical to integer subsetting but doesnt' really do anything here. But it can make a difference:
    - When there is an `NA` in the logical vector, logical subsetting replaces them with `NA`, but `which()` drops these values.
    - `x[-which(y)]` is **not** equivalent to `x[!y]`
- Bottom line: avoid switching from logical to integer subsetting unless you really want to, like to find the first (last) `TRUE` value.