Data Science with R

Part V: Subsetting and Positioning

Raphael Schleutker

March 20, 2020

Walking on water and developing software from a specification are easy if both are frozen.
— Edward V. Berard

Table of contents

A primer on logical values

A primer on logical values

Logical values appear very simple since they only take one of two possible values. But this simplicity constitute the beauty of logical values.

```
x <- sample(14)
x

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

A primer on logical values

Usually, logical vectors are created by comparison.

```
x < 10
```

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

```
x <= 10
```

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

```
x == 10
```

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

A primer on logical values

```
x != 10


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

A primer on logical values

We can invert logical values by using `!`. It is called negation operator.

```
x != 10

#  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#  [8]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE

!(x == 10)       # The same as above.

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

As for mathematical operators we can use parentheses to define the precedence of operators.

A primer on logical values

We can always identify pairs of opposed operators.

```
x <= 10

#  [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
#  [8]  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE

!(x > 10)

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

Both are completely equivalent.

A primer on logical values

Sometimes, we want to check if the elements of one vector are among the elements in another vector. We can use `%in%` for this.

```
x %in% c(1,4,17)

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

For each element in `x` `TRUE` is returned if it is equal to any element of the right-hand vector.

A primer on logical values

Comparisons are one kind of logical operators. The other kind are connective operators. They take two logical values and return a logical value.

```
y <- x < 10
z <- x > 10
```

A primer on logical values

```
y & z   # Intersection, conjunction

#  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#  [8] FALSE FALSE FALSE FALSE FALSE FALSE FALSE

y | z   # Union, disjunction

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

Working with logical values is closely related to set theory and there are some mathematical rules that can be applied. However, a basic and intuitive understanding is usually all we need.

A primer on logical values

```
y & z   # Intersection, conjunction

#  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#  [8] FALSE FALSE FALSE FALSE FALSE FALSE FALSE

y | z   # Union, disjunction

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

Working with logical values is closely related to set theory and there are some mathematical rules that can be applied. However, a basic and intuitive understanding is usually all we need.

A primer on logical values

Remember the %in% operator? We can express the very same with unions:

```
x %in% c(1,4,17)

#  [1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
#  [8] FALSE FALSE FALSE FALSE FALSE FALSE FALSE

x == 1 | x == 4 | x == 17 # Less convenient but equivalent.

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

Vectors

Vectors

We have seen so far how we can use vectors and lists/data frame to store our information and we have seen how we can get back the stored information by calling the variable name. Often, we only want specific values of a vector or list.

We can extract specific values by different methods:

- By its position in the vector.
- By its name.
- By its value.

Let's create a vector with shuffled values from 1 to 100.

```
a <- sample(100)
```

Vectors – Subsetting by position

Vectors are ordered, meaning that the first element always stays the first element as long as we do not change this actively. So we can refer to an element in a vector by its position. For this, we use the [operator (which is also a function: It does something!).

```
a[3]


# [1] 93
```

Vectors – Subsetting by position

We know that single values are also vectors (of length one). So we could try to use vectors of longer length.

```
a[c(3,6,7)]
```

```
# [1] 93  2 91
```

Apparently, we can also use vectors of arbitrary length. We can even use values that exceed the length of the vector.

```
a[101]
```

```
# [1] NA
```

Vectors – Subsetting by position

We can also extract values several times from a vector.

```
a[c(3,3,4,4)]


# [1] 93 93 25 25
```

Vectors – Subsetting by position

We have seen how we can can create special patterns of values using `seq` and `rep`. This comes in handy for subsetting by position. For instance, if we want to extract every second element.

```
seq(1, length(a), 2)
```

```
#  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29
# [16] 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59
# [31] 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89
# [46] 91 93 95 97 99
```

```
a[seq(1, length(a), 2)]
```

```
#  [1] 92 93 86 91 73 56 16 81 94 57  4 77 66  6 17
# [16] 12 55 14 65 79 40 35 82 11 10 22 70  7 33 58
# [31] 95 83 38 60 72 48 24 23 64 98 54 62 18 51 19
# [46] 78 74  8 50 87
```

Vectors – Subsetting by position

Sometimes we want to get all values except some specific one. In this case we can use negative values.

```
-seq(1, length(a), 5)
```

```
#  [1]  -1  -6 -11 -16 -21 -26 -31 -36 -41 -46 -51
# [12] -56 -61 -66 -71 -76 -81 -86 -91 -96
```

```
a[-seq(1, length(a), 5)]
```

```
#  [1]  20  93  25  86  91  41  73  31  75  16  89
# [12]  81  94  46  57  28  21  77  26  66   6  59
# [23]  17  53  27  55 100  14  65  13  79  90   5
# [34]  35  32  82  11  29  10  39  76  70  43   7
# [45]  33  30  58  15  68  83   1  38  60  36  72
# [56]  45  49  24  63  23  64  42  98  71  84  62
# [67]  67  18  51  99  19  97  52  74  69   8  50
# [78]   3  87   9
```

Vectors – Subsetting by name

As for lists and data frame we can give elements of vectors names. We can do so either directly when we create the vector or we do it later by using `names`

```
c(a = 1, b = 2, c = 3)

# a b c
# 1 2 3

b <- 1:26
names(b) <- letters
b

#  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
#  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
#  q  r  s  t  u  v  w  x  y  z
# 17 18 19 20 21 22 23 24 25 26
```

Vectors – Subsetting by name

Of course we can still use subsetting by position for named vectors.

```
b[1:10]


#  a  b  c  d  e  f  g  h  i  j
#  1  2  3  4  5  6  7  8  9 10
```

But now we can also refer to elements by their names. Instead of a number we add a character string with the name.

```
b["d"]


# d
# 4
```

Vectors – Subsetting by name

Of course we can use vectors with several names.

```
b[c("a", "a", "b", "c")]


# a a b c
# 1 1 2 3
```

Vectors – Double brackets for vectors

For subsetting by position and name we can either use single brackets as shown so far, or double brackets `[[`. Double brackets only take a single value and accordingly we can only extract a single value at a time. The main difference for vectors is, that double brackets drop the name of the element.

```
b[1]

# a
# 1

b[[1]]

# [1] 1
```

Vectors – Logical subsetting

The third way to extract specific values is logical subsetting or subsetting by value. We can state for each position if we want to extract it.

```
c <- 1:5
c[c(TRUE, TRUE, FALSE, FALSE, TRUE)]

# [1] 1 2 5
```

Vectors – Logical subsetting

Logical vectors that are shorter than the vector from which we want to extract values will be recycled. So, another way to extract every second element is

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


#  [1] 92 93 86 91 73 56 16 81 94 57  4 77 66  6 17
# [16] 12 55 14 65 79 40 35 82 11 10 22 70  7 33 58
# [31] 95 83 38 60 72 48 24 23 64 98 54 62 18 51 19
# [46] 78 74  8 50 87
```

A general rule in programming is that a goal can be achieved in several ways. Some ways are obviously more elegant than others but often, it is simply a matter of taste.

(I, personally, like the above solution :))

Vectors – Logical subsetting

Comparisons create logical vectors. We can utilize this to extract elements according to their value.

```
d <- a < 25
head(d)

# [1] FALSE  TRUE FALSE FALSE FALSE  TRUE

length(d)

# [1] 100

a[d]

#  [1] 20  2 16  4 21  6 17 12 14 13  5 11 10 22  7
# [16] 15  1 24 23 18 19  8  3  9
```

Vectors – Logical subsetting

Usually, we do not store the logical vector in a variable but use it directly.

```
a[a < 25]


#  [1] 20  2 16  4 21  6 17 12 14 13  5 11 10 22  7
# [16] 15  1 24 23 18 19  8  3  9
```

Vectors – Logical subsetting

Sometimes we are not interested in the value for which a comparisons is `TRUE` but in its position in the vector. We can use `which` for this. This function takes a logical vector and returns all positions of `TRUE`.

```
which(a < 25)

#  [1]   2   6  13  21  22  27  29  31  35  38  42
# [12]  47  49  51  55  60  64  73  75  85  89  95
# [23]  98 100
```

A common case is to find the position of the smallest or largest value in a vector. For this, we can use `which.min` and `which.max`.

```
which.min(a)

# [1] 64
```

Vectors – Logical subsetting

If we want to find the position of several values in a vector, we can use `match`.

```
match(1:10, a)

#  [1]  64   6  98  21  42  27  55  95 100  49
```

For each element in the first element, `match` returns the position of the first appearance of that value in the second vector.

Vectors – Modifying vectors

Interestingly, we can use all of the above methods not only to extract values but also to modify them.

```
a[a < 25] <- 0
a
```

```
#   [1]  92   0  93  25  86   0  91  41  73  31  56
#  [12]  75   0  89  81  61  94  46  57  28   0   0
#  [23]  77  26  66  34   0  59   0  53   0  27  55
#  [34] 100   0  96  65   0  79  90  40   0  35  32
#  [45]  82  80   0  29   0  39   0  76  70  43   0
#  [56]  88  33  30  58   0  95  68  83   0  38  44
#  [67]  60  36  72  45  48  49   0  63   0  37  64
#  [78]  42  98  71  54  84  62  67   0  47  51  99
#  [89]   0  97  78  52  74  69   0  85  50   0  87
# [100]   0
```

Vectors – Modifying vectors

This can be, for instance, used to re-order a vector.

```
e <- c(2, 5, 3, 8, 9, 1)
e[c(2,3)] <- e[c(3,2)]  # Switch 2nd and 3rd element.
e


# [1] 2 3 5 8 9 1
```

Lists

Lists

We can use the very same mechanisms to extract elements from lists.

```
l1 <- list(a = 1:10, b = list(100:90, letters[1:10]), c = mean)
```

Lists

```
l1[1]

# $a
#  [1]  1  2  3  4  5  6  7  8  9 10

l1["a"]

# $a
#  [1]  1  2  3  4  5  6  7  8  9 10

l1[c(TRUE, FALSE, FALSE)]

# $a
#  [1]  1  2  3  4  5  6  7  8  9 10
```

Lists – Single and double brackets

For lists, single brackets always return a list, also for single elements.

```
typeof(l1[1])

# [1] "list"
```

Double brackets return the actual element that is stored at the position without an enclosing list.

```
typeof(l1[[1]])

# [1] "integer"
```

Lists – Single and double brackets

```
l1[1]

# $a
#  [1]  1  2  3  4  5  6  7  8  9 10

l1[[1]]

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

Lists – Subsetting by name

In case of named lists, we have an additional way to retrieve an element by name, i.e. by using the $ operator.

```
l1$a
```

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

```
l1[["a"]]
```

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

Both are completely equivalent. Using brackets however allows us to retrieve elements with computed values whereas for $ we always have to hard-code the name ourselves.

Lists – Recursive subsetting

[[returns the element at a specific position. If this is another list or a vector we can directly subset the result again.

```
l1[[2]]

# [[1]]
#  [1] 100  99  98  97  96  95  94  93  92  91  90
#
# [[2]]
#  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

l1[[2]][[1]]

#  [1] 100  99  98  97  96  95  94  93  92  91  90

l1[[2]][[1]][5]

# [1] 96
```

Data frames

Data frames

We have seen that data frames are a special kind of list that are used to store tabular data. Therefore, we can use the very same techniques to retrieve elements from data frames as for lists. But data frames allow us to subset data in a more convenient way. Let's take the `iris` dataset as an example.

```
head(iris)


#   Sepal.Length Sepal.Width Petal.Length
# 1          5.1         3.5          1.4
# 2          4.9         3.0          1.4
# 3          4.7         3.2          1.3
# 4          4.6         3.1          1.5
# 5          5.0         3.6          1.4
# 6          5.4         3.9          1.7
#   Petal.Width Species
# 1         0.2  setosa
# 2         0.2  setosa
# 3         0.2  setosa
# 4         0.2  setosa
# 5         0.2  setosa
```

Data frames

An element in a table is defined by its row and column. Accordingly, we can retrieve a value by providing both values.

```
iris[1, 2]


# [1] 3.5
```

This returns the element in the first row and second column. Please notice, that we provide two arguments here, not one argument with two elements!

Data frames

Leaving away one value returns all values for that dimension.

```
iris[1,]        # First row.

#   Sepal.Length Sepal.Width Petal.Length
# 1          5.1         3.5          1.4
#   Petal.Width Species
# 1         0.2  setosa

iris[,1]        # First column.

#   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4
#  [12] 4.8 4.8 4.3 5.8 5.7 5.4 5.1 5.7 5.1 5.4 5.1
#  [23] 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2
#  [34] 5.5 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0
#  [45] 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5 6.5
#  [56] 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7
#  [67] 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8
#  [78] 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3
```

Data frames

Of course, we can also use more than one value for each dimension.

```
iris[c(2,3), c(4,5)]

#   Petal.Width Species
# 2         0.2  setosa
# 3         0.2  setosa
```

There is no way to retrieve several single elements. So we can not get the element in the second row and fourth column and the element in the third row and fifth column in the same call.

Data frames – Logical subsetting

Logical subsetting is also possible for data frames but usually a bit inconvenient.

```
iris[iris$Sepal.Length > 4.5,]
```

```
#     Sepal.Length Sepal.Width Petal.Length
# 1            5.1         3.5          1.4
# 2            4.9         3.0          1.4
# 3            4.7         3.2          1.3
# 4            4.6         3.1          1.5
# 5            5.0         3.6          1.4
# 6            5.4         3.9          1.7
# 7            4.6         3.4          1.4
# 8            5.0         3.4          1.5
# 10           4.9         3.1          1.5
# 11           5.4         3.7          1.5
# 12           4.8         3.4          1.6
# 13           4.8         3.0          1.4
# 15           5.8         4.0          1.2
```

Data frames – subset

A useful function for data frames is `subset`. As the name suggests, it returns a subset of the data frame.

```
subset(iris, Sepal.Length > 4.5, select = Petal.Length)
```

```
#     Petal.Length
# 1            1.4
# 2            1.4
# 3            1.3
# 4            1.5
# 5            1.4
# 6            1.7
# 7            1.4
# 8            1.5
# 10           1.5
# 11           1.5
# 12           1.6
# 13           1.4
# 15           1.2
```