
(http://software-carpentry.org)

# Subsetting data

R has many powerful subset operators and mastering them will allow you to easily perform complex operation on any kind of dataset. Allows you to manipulate data very succinctly.

As the last section for this topic we'll cover:

- The three subsetting operators,
- The six types of subsetting,
- Important difference in subsetting behaviour for different objects
- Using subsetting in conjunction with assignment

# Subsetting atomic vectors

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
```

**We can subset this in many ways**

1. Using positive integers

```
x[1]
x[c(3,1)]
# We can duplicate indices
x[c(1, 1)]
# Real numbers are silently truncated to integers
x[c(2.1, 2.9)]
```

1. Using negative integers

```
# skip the first element
x[-1]
# skip the first and the fifth
x[-c(1,5)]
```

1. Using logical operators

```
x[c(TRUE, TRUE, FALSE, FALSE)]
# or based on a condition
x[x > 3]
x[which(x >3)]
Also see `which.max()` and `which.min()`
```

1. with nothing

```
x[]
# useful when dealing with 2 or higher dimensional objects
```

1. with zero

```
x[0]
# helpful for generating test data or creating empty objects
```

1. Referencing objects by their names

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

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

# We can also repeat indices
y[c("a", "a", "a")]

# Names are always matched exactly, not partially
z <- c(abc = 1, def = 2)
z[c("a", "d")]
# <NA> <NA>
#   NA   NA
```

---

# Subsetting lists

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

```
x <- as.list(1:10)

x[1:5]

# What class is this object?
```

To extract individual elements inside a list, use the `[[` operator

```
# to get element 5

x[[5]]

class(x[[5]])

another_variable <- x[[5]]
```

Or using their names

```
names(x) <- letters[1:5]

x$a
x[["a"]]
```

# Subsetting matrices

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

When you extract a single column or row, you get a vector out. This is not always what is wanted, so the `drop` argument comes in useful

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

# Subsetting data frames

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

df[df$x == 2, ]
#   x y z
# 2 2 2 b
df[c(1, 3), ]
#   x y z
# 1 1 3 a
# 3 3 1 c

# There are two ways to select columns from a data frame
# Like a list:
df[c("x", "z")]
#   x z
# 1 1 a
# 2 2 b
# 3 3 c
# Like a matrix
df[, c("x", "z")]
#   x z
# 1 1 a
# 2 2 b
# 3 3 c

# There's an important difference if you select a simple column:
# matrix subsetting simplifies by default, list subsetting does not.
df["x"]
#   x
# 1 1
# 2 2
# 3 3
df[, "x"]
# [1] 1 2 3
```

If you use the single bracket subset you get a new dataframe out. If you use `$` or `[[` you get the contents of that column. If the column does not exist, `$` will return `NULL` and `[[` will return an error.

```
v <- "x"
df[[v]]
df$v # this does not work!
df$x # this does
df$not_in_here # NULL
df[["not_in_here"]] # error
```

---

**Facebook (https://www.facebook.com/SoftwareCarpentry)**
**Google+ (https://plus.google.com/u/0/114244759874490019250/posts)**