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.

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

- **Positive integers** return elements at the specified positions:

  ```
  x[c(3, 1)]
  #> [1] 3.3 2.1
  x[order(x)]
  #> [1] 2.1 3.3 4.2 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
  ```

- **Negative integers** omit elements at the specified positions:

  ```
  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
  ```

- **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(TRUE, TRUE, FALSE, FALSE)]
  #> [1] 2.1 4.2
  x[x > 3]
  #> [1] 4.2 3.3 5.4
  ```

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

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

```
#> [1] 2.1 3.3
# Equivalent to
x[c(TRUE, FALSE, TRUE, FALSE)]
#> [1] 2.1 3.3
```

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
```

- **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)
```

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

# 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 are always matched exactly
z <- c(abc = 1, def = 2)
z[c("a", "d")]
#> <NA> <NA>
#>   NA   NA
```

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.

Matrices and arrays

You can subset higher-dimensional structures in three ways:

- With multiple vectors.
- With a single vector.
- With a matrix.

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.

```
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
```

By default, `[` will simplify the results to the lowest possible dimensionality. Seesimplifying vs. preserving to learn how to avoid this.

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 <- outer(1:5, 1:5, FUN = "paste", sep = ","))
#>      [,1]  [,2]  [,3]  [,4]  [,5]
#> [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
#> [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
#> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
#> [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
#> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"
vals[c(4, 15)]
#> [1] "4,1" "5,3"
```

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:

```
vals <- outer(1:5, 1:5, FUN = "paste", sep = ",")
select <- matrix(ncol = 2, byrow = TRUE, c(
  1, 1,
  3, 1,
  2, 4
))
vals[select]
#> [1] "1,1" "3,1" "2,4"
```

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 = 1:3, 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 single
# column: matrix subsetting simplifies by default, list
# subsetting does not.
```

```
str(df["x"])
#> 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
str(df[, "x"])
#>  int [1:3] 1 2 3
```

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.

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 it can return only a single value, you must use `[[` with either a single positive integer or a string:

```
a <- list(a = 1, b = 2)
a[[1]]
#> [1] 1
a["a"]]
#> [1] 1

# If you do supply a vector it indexes recursively
b <- list(a = list(b = list(c = list(d = 1))))
b[[c("a", "b", "c", "d")]]
#> [1] 1
# Same as
b["a"]]["b"]]["c"]]["d"]]
#> [1] 1
```

Because data frames are lists of columns, you can use `[[` to extract a column from data frames: `mtcars[[1]]`, `mtcars[["cyl"]]`.

S3 and S4 objects can override the standard behaviour of `[` and `[[` so they behave differently for different types of objects. The key difference is usually how you select between simplifying or preserving behaviours, and what the default is.

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]`
Array	`x[1, ]` **or** `x[, 1]`	`x[1, , drop = F]` **or** `x[, 1, drop = F]`
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[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)
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** or **array**: if any of the dimensions has length 1, drops that dimension.

```
a <- matrix(1:4, nrow = 2)
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 = 1:2)
str(df[1])
#> 'data.frame':    2 obs. of  1 variable:
#>  $ a: int  1 2
str(df[[1]])
#>  int [1:2] 1 2
str(df[, "a", drop = FALSE])
#> 'data.frame':    2 obs. of  1 variable:
#>  $ a: int  1 2
str(df[, "a"])
#>  int [1:2] 1 2
```

$

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

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
```

There's one important difference between `$` and `[[`. `$` does partial matching:

```
x <- list(abc = 1)
x$a
#> [1] 1
x[["a"]]
#> NULL
```

If you want to avoid this behaviour you can set the global option `warnPartialMatchDollar` to `TRUE`. Use with caution: it may affect behaviour in other code you have loaded (e.g., from a package).

Missing/out of bounds indices

`[` and `[[` differ slightly in their behaviour when the index is out of bounds (OOB), for example, when you try to extract the fifth element of a length four vector, or subset a vector with `NA` or `NULL`:

```
x <- 1:4
str(x[5])
#>  int NA
str(x[NA_real_])
#>  int NA
str(x[NULL])
#>  int(0)
```

The following table summarises the results of subsetting atomic vectors and lists with `[` and `[[` and different types of OOB value.

Operator	Index	Atomic	List
`[`	OOB	`NA`	`list(NULL)`
`[`	`NA_real_`	`NA`	`list(NULL)`
`[`	`NULL`	`x[0]`	`list(NULL)`
`[[`	OOB	Error	Error
`[[`	`NA_real_`	Error	`NULL`
`[[`	`NULL`	Error	Error

If the input vector is named, then the names of OOB, missing, or `NULL` components will be `"<NA>"`.

Subsetting and assignment

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

```
x <- 1:5
x[c(1, 2)] <- 2:3
x
#> [1] 2 3 3 4 5

# The length of the LHS needs to match the RHS
x[-1] <- 4:1
x
#> [1] 2 4 3 2 1

# Note that there's no checking for duplicate indices
x[c(1, 1)] <- 2:3
x
#> [1] 3 4 3 2 1

# You can't combine integer indices with NA
x[c(1, NA)] <- c(1, 2)
#> Error in x[c(1, NA)] <- c(1, 2): NAs are not allowed in subscripted assignments
# But you can combine logical indices with NA
# (where they're treated as false).
x[c(T, F, NA)] <- 1
x
#> [1] 1 4 3 1 1

# This is mostly useful when conditionally modifying vectors
df <- data.frame(a = c(1, 10, NA))
df$a[df$a < 5] <- 0
df$a
#> [1]  0 10 NA
```

Subsetting with nothing can be useful in conjunction with assignment because it will preserve the original object class and structure. Compare the following two expressions. In the first, `mtcars` will remain as a data frame. In the second, `mtcars`will become a list.

```
mtcars[] <- lapply(mtcars, as.integer)
mtcars <- lapply(mtcars, as.integer)
```

With lists, you can use subsetting + assignment + `NULL` to remove components from a list. To add a literal `NULL` to a list, use `[` and `list(NULL)`:

```
x <- list(a = 1, b = 2)
x[["b"]] <- NULL
str(x)
#> List of 1
#>  $ a: num 1

y <- list(a = 1)
y["b"] <- list(NULL)
str(y)
#> List of 2
#>  $ a: num 1
#>  $ b: NULL
```

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.

Lookup tables (character subsetting)

Character matching provides a powerful way to make lookup tables. Say you want to convert abbreviations:

```
x <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "Male", f = "Female", u = NA)
lookup[x]
#>        m        f        u        f        f        m        m
#>   "Male" "Female"       NA "Female" "Female"   "Male"   "Male"
unname(lookup[x])
#> [1] "Male"   "Female" NA       "Female" "Female" "Male"   "Male"

# Or with fewer output values
c(m = "Known", f = "Known", u = "Unknown")[x]
#>         m         f         u         f         f         m         m
#>   "Known"   "Known" "Unknown"   "Known"   "Known"   "Known"   "Known"
```

If you don't want names in the result, use `unname()` to remove them.

Matching and merging by hand (integer subsetting)

You may have a more complicated lookup table which has multiple columns of information. Suppose we have a vector of integer grades, and a table that describes their properties:

```
grades <- c(1, 2, 2, 3, 1)

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

We want to duplicate the info table so that we have a row for each value in `grades`. We can do this in two ways, either using `match()` and integer subsetting, or `rownames()` and character subsetting:

```
grades
#> [1] 1 2 2 3 1

# Using match
id <- match(grades, info$grade)
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

# Using rownames
rownames(info) <- info$grade
info[as.character(grades), ]
#>     grade      desc  fail
#> 1       1      Poor  TRUE
#> 2       2      Good FALSE
#> 2.1     2      Good FALSE
#> 3       3 Excellent FALSE
#> 1.1     1      Poor  TRUE
```

If you have multiple columns to match on, you'll need to first collapse them to a single column (with `interaction()`, `paste()`, or `plyr::id()`). You can also use `merge()` or `plyr::join()`, which do the same thing for you — read the source code to see how.

Random samples/bootstrap (integer subsetting)

You can use integer indices to perform random sampling or bootstrapping of a vector or data frame. `sample()` generates a vector of indices, then subsetting to access the values:

```
df <- data.frame(x = rep(1:3, each = 2), y = 6:1, z = letters[1:6])
```

```
# Set seed for reproducibility
set.seed(10)

# Randomly reorder
df[sample(nrow(df)), ]
#>   x y z
#> 4 2 3 d
#> 2 1 5 b
#> 5 3 2 e
#> 3 2 4 c
#> 1 1 6 a
#> 6 3 1 f
# Select 3 random rows
df[sample(nrow(df), 3), ]
#>   x y z
#> 2 1 5 b
#> 6 3 1 f
#> 3 2 4 c
# Select 6 bootstrap replicates
df[sample(nrow(df), 6, rep = T), ]
#>     x y z
#> 3   2 4 c
#> 4   2 3 d
#> 4.1 2 3 d
#> 1   1 6 a
#> 4.2 2 3 d
#> 3.1 2 4 c
```

The arguments of `sample()` control the number of samples to extract, and whether sampling is performed with or without replacement.

Ordering (integer subsetting)

`order()` 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"
```

To break ties, you can supply additional variables to `order()`, and you can change from ascending to descending order using `decreasing = TRUE`. By default, any missing values will be put at the end of the vector; however, you can remove them with `na.last = NA` or put at the front with `na.last = FALSE`.

For two or more dimensions, `order()` and integer subsetting makes it easy to order either the rows or columns of an object:

```
# Randomly reorder df
df2 <- df[sample(nrow(df)), 3:1]
df2
#>   z y x
#> 3 c 4 2
#> 1 a 6 1
#> 2 b 5 1
#> 4 d 3 2
#> 6 f 1 3
#> 5 e 2 3

df2[order(df2$x), ]
#>   z y x
#> 1 a 6 1
#> 2 b 5 1
#> 3 c 4 2
#> 4 d 3 2
#> 6 f 1 3
#> 5 e 2 3
df2[, order(names(df2))]
#>   x y z
#> 3 2 4 c
#> 1 1 6 a
#> 2 1 5 b
#> 4 2 3 d
#> 6 3 1 f
#> 5 3 2 e
```

More concise, but less flexible, functions are available for sorting vectors, `sort()`, and data frames, `plyr::arrange()`.

Expanding aggregated counts (integer subsetting)

Sometimes you get a data frame where identical rows have been collapsed into one and a count column has been added. `rep()` and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index:

```
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
```

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:

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

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

```
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
```

If you know the columns you don't want, use set operations to work out which colums to keep:

```
df[setdiff(names(df), "z")]
#>   x y
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

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.

```
mtcars[mtcars$gear == 5, ]
#>     mpg cyl  disp  hp drat    wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
#> 29 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
#> 30 19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
#> 31 15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
```

```
mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
#>     mpg cyl  disp  hp drat    wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
```

Remember to use the vector boolean operators `&` and `|`, not the short-circuiting scalar operators `&&` and `||` which are more useful inside if statements. Don't forgetDe Morgan's laws, which can be useful to simplify negations:

- `!(X & Y)` is the same as `!X | !Y`
- `!(X | Y)` is the same as `!X & !Y`

For example, `!(X & !(Y | Z))` simplifies to `!X | !!(Y|Z)`, and then to `!X | Y | Z`.

`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. You'll learn how it works in non-standard evaluation.

```
subset(mtcars, gear == 5)
#>     mpg cyl  disp  hp drat    wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
#> 29 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
#> 30 19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
#> 31 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
#> 27 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
```

Boolean algebra vs. sets (logical & integer subsetting)

It's useful to be aware of the natural equivalence between set operations (integer subsetting) and boolean algebra (logical subsetting). Using set operations is more effective when:

- You want to find the first (or last) `TRUE`.
- You have very few `TRUE`s and very many `FALSE`s; a set representation may be faster and require less storage.

`which()` allows you to convert a boolean representation to an integer representation. There's no reverse operation in base R but we can easily create one:

```
x <- sample(10) < 4
which(x)
#> [1]  3  7 10

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

```
  out
}
unwhich(which(x), 10)
#>  [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

Let's create two logical vectors and their integer equivalents and then explore the relationship between boolean and set operations.

```
(x1 <- 1:10 %% 2 == 0)
#>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
(x2 <- which(x1))
#> [1]  2  4  6  8 10
(y1 <- 1:10 %% 5 == 0)
#>  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
(y2 <- which(y1))
#> [1]  5 10

# X & Y <-> intersect(x, y)
x1 & y1
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
intersect(x2, y2)
#> [1] 10

# X | Y <-> union(x, y)
x1 | y1
#>  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
union(x2, y2)
#> [1]  2  4  6  8 10  5

# X & !Y <-> setdiff(x, y)
x1 & !y1
#>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
setdiff(x2, y2)
#> [1] 2 4 6 8

# xor(X, Y) <-> setdiff(union(x, y), intersect(x, y))
xor(x1, y1)
#>  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
setdiff(union(x2, y2), intersect(x2, y2))
#> [1] 2 4 6 8 5
```

When first learning subsetting, a common mistake is to use `x[which(y)]` instead of`x[y]`. Here the `which()` achieves nothing: it switches from logical to integer subsetting but the result will be exactly the same. Also beware that `x[-which(y)]` is**not** equivalent to `x[!y]`: if `y` is all FALSE, `which(y)` will be `integer(0)` and `-integer(0)` is still `integer(0)`, so you'll get no values, instead of all values. In general, avoid switching from logical to integer subsetting unless you

want, for example, the first or last `TRUE` value.