9 Subsetting R Objects

Watch a video of this section

There are three operators that can be used to extract subsets of R objects.

- The `[` operator always returns an object of the same class as the original. It can be used to select multiple elements of an object
- The `[[` operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- The `$` operator is used to extract elements of a list or data frame by literal name. Its semantics are similar to that of `[[` .

9.1 Subsetting a Vector

Vectors are basic objects in R and they can be subsetted using the `[` operator.

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]    ## Extract the first element
[1] "a"
> x[2]    ## Extract the second element
[1] "b"
```

The `[` operator can be used to extract multiple elements of a vector by passing the operator an integer sequence. Here we extract the first four elements of the vector.

```
> x[1:4]
[1] "a" "b" "c" "c"
```

The sequence does not have to be in order; you can specify any arbitrary integer vector.

```
> x[c(1, 3, 4)]
[1] "a" "c" "c"
```

We can also pass a logical sequence to the `[` operator to extract elements of a vector that satisfy a given condition. For example, here we want the elements of `x` that come lexicographically ***after*** the letter "a".

```
> u <- x > "a"
> u
[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
```

Another, more compact, way to do this would be to skip the creation of a logical vector and just subset the vector directly with the logical expression.

```
> x[x > "a"]
[1] "b" "c" "c" "d"
```

9.2 Subsetting a Matrix

Watch a video of this section

Matrices can be subsetted in the usual way with (***i,j***) type indices. Here, we create simple \(2\times3\) matrix with the `matrix` function.

```
> x <- matrix(1:6, 2, 3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

We can access the \((1,2)\) or the \((2,1)\) element of this matrix using the appropriate indices.

```
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

Indices can also be missing. This behavior is used to access entire rows or columns of a matrix.

```
> x[1, ]  ## Extract the first row
[1] 1 3 5
> x[, 2]  ## Extract the second column
[1] 3 4
```

9.2.1 Dropping matrix dimensions

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a \(1\times1\) matrix. Often, this is exactly what we want, but this behavior can be turned off by setting `drop = FALSE`.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop = FALSE]
     [,1]
[1,]    3
```

The result is a matrix

Similarly, when we extract a single row or column of a matrix, R by default drops the dimension of length 1, so instead of getting a \(1\times3\) matrix after extracting the first row, we get a vector of length 3. This behavior can similarly be turned off with the `drop = FALSE` option.

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, , drop = FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

matrix

Be careful of R's automatic dropping of dimensions. This is a feature that is often quite useful during interactive work, but can later come back to bite you when you are writing longer programs or functions.

9.3 Subsetting Lists

Watch a video of this section

Lists in R can be subsetted using all three of the operators mentioned above, and all three are used for different purposes.

```
> x <- list(foo = 1:4, bar = 0.6)
> x
$foo
[1] 1 2 3 4

$bar
[1] 0.6
```

The `[[` operator can be used to extract ***single*** elements from a list. Here we extract the first element of the list.

```
> x[[1]]
[1] 1 2 3 4
```

The `[[` operator can also use named indices so that you don't have to remember the exact ordering of every element of the list. You can also use the `$` operator to extract elements by name.

```
> x[["bar"]]
[1] 0.6
> x$bar
[1] 0.6
```

Notice you don't need the quotes when you use the `$` operator.

One thing that differentiates the `[[` operator from the `$` is that the `[[` operator can be used with ***computed*** indices. The `$` operator can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
>
> ## computed index for "foo"
> x[[name]]
[1] 1 2 3 4
>
> ## element "name" doesn't exist! (but no error here)
> x$name
NULL
>
> ## element "foo" does exist
> x$foo
[1] 1 2 3 4
```

This makes sense esp since you wouldn't say x$"foo". A little weird that there's no error though.

9.4 Subsetting Nested Elements of a List

The `[[` operator can take an integer sequence if you want to extract a nested element of a list.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
>
> ## Get the 3rd element of the 1st element
> x[[c(1, 3)]]
[1] 14
>
> ## Same as above
> x[[1]][[3]]
[1] 14
>
> ## 1st element of the 2nd element
> x[[c(2, 1)]]
[1] 3.14
```

Weird! Don't do this!
It's easy to mistake this for the 1st and 3rd elements of the list.
Use x[c(1,3)] instead. See below.

Much more sensible.

9.5 Extracting Multiple Elements of a List

The `[` operator can be used to extract ***multiple*** elements from a list. For example, if you wanted to extract the first and third elements of a list, you would do the following

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> x[c(1, 3)]
$foo
[1] 1 2 3 4

$baz
[1] "hello"
```

Note that `x[c(1, 3)]` is NOT the same as `x[[c(1, 3)]]`.

Remember that the `[` operator always returns an object of the same class as the original. Since the original object was a list, the `[` operator returns a list. In the above code, we returned a list with two elements (the first and the third).

9.6 Partial Matching

Watch a video of this section

Partial matching of names is allowed with `[[` and `$`. This is often very useful during interactive work if the object you're working with has very long element names. You can just abbreviate those names and R will figure out what element you're referring to.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4 5
```

Interesting.

In general, this is fine for interactive work, but you shouldn't resort to partial matching if you are writing longer scripts, functions, or programs. In those cases, you should refer to the full element name if possible. That way there's no ambiguity in your code.

9.7 Removing NA Values

Watch a video of this section

A common task in data analysis is removing missing values (`NA`s).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> print(bad)
[1] FALSE FALSE  TRUE FALSE  TRUE FALSE
> x[!bad]
[1] 1 2 4 5
```

What if there are multiple R objects and you want to take the subset with no missing values in any of those objects?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
[1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE
> x[good]
[1] 1 2 4 5
> y[good]
[1] "a" "b" "d" "f"
```

You can use `complete.cases` on data frames too.

```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> good <- complete.cases(airquality)
> head(airquality[good, ])
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
7    23     299  8.6   65     5   7
8    19      99 13.8   59     5   8
```

This can be confusing though if you end up only looking at certain columns and don't care if there are NAs in excluded columns.