Modes and Data Structures

What is a “mode”?

In `R`, individual elements (the smallest unit that we store) have a “mode”, which describes the type of quantity they describe. Possible modes include:

- Integer
- Numeric (Floating Point, Double)
- Character (String)
- Logical (Boolean)
- Complex

We usually don’t need to tell `R` what the mode should be when we define something. It guesses from what we provide.

We can use the `str` and `typeof` functions to learn what the mode of a variable we have defined is.

Numeric (Floating Point, Double)

```r
x <- 1.1
str(x)
```

```
 num 1.1
```

```r
typeof(x)
```

```
[1] "double"
```

```r
mode(x)
```

```
[1] "numeric"
```

Character (String)

```r
x <- "a"
str(x)
```

```
 chr "a"
```

```r
typeof(x)
```

```
[1] "character"
```

```r
mode(x)
```

```
[1] "character"
```

Note: Characters can include more than one element, e.g. `x <- "abc"`.

Logical (Boolean)

```r
x <- TRUE
str(x)
```

```
 logi TRUE
```

```r
typeof(x)
```

```
[1] "logical"
```

```r
mode(x)
```

```
[1] "logical"
```

The logical mode can take on values `TRUE` and `FALSE`, which can be abbreviated `T` and `F`.

Note: For this reason, naming variables `T` or `F` is discouraged.

Integer?

```r
x <- 1
str(x)
```

```
 num 1
```

```r
typeof(x)
```

```
[1] "double"
```

```r
mode(x)
```

```
[1] "numeric"
```

If `R` has to guess whether a number is an integer or a numeric, it will default to numeric.

Integer!

We actually do need to tell R the mode when we want to define an integer. A way to do that is to apply the function `as.integer` to the integer we provide.

```r
x <- as.integer(1)
str(x)
```

```
 int 1
```

```r
typeof(x)
```

```
[1] "integer"
```

```r
mode(x)
```

```
[1] "numeric"
```

Vectors

Vectors are collections of elements that share the same mode.

The length of a vector describes the number of elements in a vector.

In fact, everything we've seen so far was a vector of length 1!

```r
x <- 1
str(x)
```

```
 num 1
```

Creating a Vector

We can construct vectors from multiple elements using the c function, where c stands for **concatenate**.

```
x <- c(1, 5, 2)
```

```
str(x)
```

```
 num [1:3] 1 5 2
```

```
x
```

```
[1] 1 5 2
```

Determining the Number of Elements in a Vector

The `length` function, when applied to a vector, returns the number of elements in a vector.

```r
length(x)
```

```
[1] 3
```

Viewing an Element of a Vector

```r
x[1]
```

```
[1] 1
```

```r
x[2]
```

```
[1] 5
```

```r
x[3]
```

```
[1] 2
```

Viewing Elements of a Vector

```r
x[1:2]
```

```
[1] 1 5
```

```r
x[c(1, 3)]
```

```
[1] 1 2
```

```r
x[-2]
```

```
[1] 1 2
```

```r
x[-c(1, 3)]
```

```
[1] 5
```

Growing a Vector

Unlike some other languages, `R` allows you to make a vector longer or make it shorter.

```
x <- c(x, 4)
```

```
x
```

```
[1] 1 5 2 4
```

Shortening a Vector

```r
x <- x[1:3]
```

```r
x
```

```
[1] 1 5 2
```

Replacing an Element of a Vector

```r
x[2] <- 5.1
```

```r
x
```

```
[1] 1.0 5.1 2.0
```

Elementwise Assignment of a Vector

We can only assign a value to an individual element of a vector if the vector exists.

For instance, we have not defined `z`.

What happens if we type `z[1] <- 2` without defining `z`?

Elementwise Assignment of a Vector

Once a vector has been defined, we can assign a value to any element of the vector, even if we are assigning a value to an element of the vector that doesn't exist yet.

```r
x[4] <- 4
```

```r
x
```

```
[1] 1.0 5.1 2.0 4.0
```

```r
x[6] <- 30
```

```r
x
```

```
[1]  1.0  5.1  2.0  4.0   NA 30.0
```

What the heck is NA???

R encodes missing values as NA for all modes.

NA means there should be a value, but there isn't.

```r
c(1, NA, 3)
```

```
[1]  1 NA  3
```

R has another way of denoting that something is undefined, which is the value NULL. However, NULL means something different than NA. NULL means that the element or object does not exist at all.

```r
c(1, NULL, 3)
```

```
[1] 1 3
```

Other Ways to Create a Vector

These create "empty" length 3 vectors.

```r
x <- numeric(3)
```

```r
x <- vector(length = 3)
```

```r
x <- rep(NA, 3)
```

Alternatively, because we can grow vectors in R, initializing a vector x of a smaller size or assigning x to be NA, or NULL will work.

```r
x <- NA
```

```r
x <- NULL
```

```r
x[1] <- 1
x[2] <- 5.1
x[3] <- 2
```

Looping Over Elements of a Vector

It is common that we may want to apply a function to one element of a vector at a time.

```r
for (i in 1:length(x)) {
  x[i] <- i
}
```

```r
x
```

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

Math Operations on Numeric Vectors

Many basic mathematical operations, including addition, are performed elementwise when applied to one or more vectors in R.

```r
x <- c(1, 4)
y <- c(9.2214, 0.12)
x + y
```

```
[1] 10.2214  4.1200
```

Math Operations on Numeric Vectors

Other operations include:

- Subtraction `-`
- Multiplication `*`
- Division `/`
- Square rooting `sqrt`
- Exponentiation `^`
- Rounding `round`
- Absolute value `abs`
- Cumulative sum `cumsum`

Operations on Vectors - Recycling

```r
y <- y[1]
x + y
```

```
[1] 10.2214 13.2214
```

Although we're demonstrating recycling with addition, the same behavior appears in the context of subtraction, multiplication, and division.

Recycling also comes up when we use logical vectors to subset a vector.

Subsetting/Filtering Using Logical Vectors

Earlier we saw that we can subset a vector by providing a vector of indices that we would like to retain, e.g.

```r
x <- c(1, 5, 2)
x[c(1, 3)]
```

```
[1] 1 2
```

`x[c(1, 3)]` creates a new vector by taking a subset of elements of the original vector, specifically the 1st and third elements.

We can also subset a vector by providing a logical vector of the same length, and elements of the original vector that correspond elements of the logical vector with value `TRUE` will be retained.

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

```
[1] 1 2
```

Operations on Vectors that Yield Logical Vectors

The ability to subset vectors using logical vectors of the same length helps us subset vectors according to their values. Specifically, we have the following functions:

- Greater than >
- Greater than or equal to >=
- Less than <
- Less than or equal to <=
- Equal to ==
- Not equal to `!=`

```
x < 5
```

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

Note that these will return `NA` when applied to `NA`.

Recycling when Using Logicals to Subset

Recycling comes up in the context of subsetting/filtering a vector if we subset a vector by a shorter logical vector.

To make sense of the fact that the logical vector is shorter than the vector it is being used to subset, `R` will just repeat the logical vector over and over until it is the same length as the vector it is being used to subset.

```r
x[TRUE]
```

```
[1] 1 5 2
```

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

```
[1] 1 2
```

Creating Special Vectors - Repeating

We have now seen recycling twice. What is happening when `R` recycles a vector is that it repeats a vector to achieve a certain length.

This introduces the idea of creating a special vector that repeats itself.

Creating Special Vectors - Repeating

We can create a repeating vector using the `rep` function, which has several arguments. Two determine what should be repeated:

- The first argument is the vector that should be repeated
- The `each` argument indicates the number of times each element of the first argument should be repeated in succession

```r
rep(c("a", "b", "c"))
```

```
[1] "a" "b" "c"
```

```r
rep(c("a", "b", "c"), each = 2)
```

```
[1] "a" "a" "b" "b" "c" "c"
```

Creating Special Vectors - Repeating

Given what should be repeated, determined by the first argument to `rep` and a value of `each` (if provided), either `times` or `length.out` can be specified to determine how many times the repeating should occur:

- The `times` argument indicates how many times the repeating should occur
- The `length.out` argument indicates the length of the repeated vector

```
rep(c("a", "b", "c"), each = 2, length.out = 2)
```

```
[1] "a" "a"
```

```
rep(c("a", "b", "c"), each = 2, times = 2)
```

```
 [1] "a" "a" "b" "b" "c" "c" "a" "a" "b" "b" "c" "c"
```

Creating Special Vectors - Sequences

Frequently, we may want to create a vector with a special desired structure. For instance, we may want to make a vector with elements that are equally spaced from one minimum value up to a maximum value.

- Using `:` gives a sequence of integers, `1:5` or `0:2`
- Using `seq` gives a numeric sequence based on specified minimum and maximum values and either:
  - Specification of the number of elements between the minimum and maximum, `length.out`
  - Specification of the difference between consecutive values, `by`

```r
x <- seq(0, 5, by = 1)
x <- seq(0, 5, length.out = 6)
```

Do these two commands produce the same `x`?

Functions Summarizing Numeric Vectors

- `sum`
- `mean`
- `sd`
- `var`
- `min`
- `max`
- `prod`
- `range`

What happens if the vector contains an `NA`?

Functions Summarizing Numeric Vectors with NA

```r
x <- c(1, 3, NA)
```

```r
mean(x)
```

```
[1] NA
```

```r
mean(x, na.rm = TRUE)
```

```
[1] 2
```

What is `na.rm = TRUE` doing?

```r
mean(x, na.rm = TRUE)
```

```
[1] 2
```

```r
x
```

```
[1]  1  3 NA
```

```r
na.omit(x)
```

```
[1] 1 3
attr(,"na.action")
[1] 3
attr(,"class")
[1] "omit"
```

```r
mean(na.omit(x))
```

```
[1] 2
```

What is `na.rm = TRUE` doing?

```r
mean(x, na.rm = TRUE)
```

```
[1] 2
```

```r
mean(x[!is.na(x)])
```

```
[1] 2
```

The function `is.na` takes a vector that may contain elements with value `NA` and returns a logical/Boolean vector with elements that are `TRUE` if the corresponding element of the provided vector is `NA` and `FALSE` otherwise.

Functions Summarizing Logical Vectors

- `all`
- `any`

These functions can behave strangely when applied to logical vectors that include `NA`.

```
any(x == 2)
```

```
[1] NA
```

```
any(x == 3)
```

```
[1] TRUE
```

Subsetting/Filtering with NA's

Earlier, we saw examples of subsetting/filtering a vector by specifying which indices we do or do not want to retain, and subsetting using logical vectors. There are two nice additional ways, which automatically deal with NA's when they are present.

```r
x[x > 1]
```

```
[1]  3 NA
```

which takes a logical vector and returns a vector of integers that correspond to the indices for which the logical vector is equal to TRUE

```r
x[which(x > 1)]
```

```
[1] 3
```

```r
which(x > 1)
```

```
[1] 2
```

Subsetting/Filtering with `NA`'s

`subset` takes two arguments, an arbitrary vector and a logical vector of the same length

```r
subset(x, x > 1)
```

```
[1] 3
```

`which` and `subset` provide more concise alternatives to another approach which combines two logical vectors

```r
x[!is.na(x) & x > 1]
```

```
[1] 3
```

Additional Useful Shortcuts

`which.max` and `which.min` can be applied to a vector and return the index of the maximum or minimum value

```r
which.max(x)
```

```
[1] 2
```

```r
which.min(x)
```

```
[1] 1
```

Comparing Vectors

- `identical` takes two vectors and returns a logical that indicates whether or not the two vectors are identical
- `all.equal` takes two vectors and returns a logical that indicates whether or not the two vectors are identical but with a little numerical wiggle room

The `ifelse` function

`ifelse` is a function that can be applied to a logical vector and allows us to specify a pair of vectors that describe what each element's value should be depending on whether the logical vector evaluates to `TRUE` or `FALSE`

```r
ifelse(x > 1, "a", "b")
```

```
[1] "b" "a" NA
```

```r
ifelse(x > 1, 1, x)
```

```
[1]  1  1 NA
```

Names for Vectors

R lets us attach a vector of names for elements to each vector, which can then be used to index elements of a vector

```
names(x) <- c("first", "second", "third")
```

```
x["second"]
```

```
second
     3
```

Matrices

Like vectors, matrices are collections of elements that share the same mode. However, they have two dimensions (rows and columns) and accordingly, are indexed by pairs of indices.

The dimensions of a matrix describe the number of rows and columns.

Making a Matrix

Making a matrix is a bit different from making a vector. We **cannot** grow a matrix in the same way we can grow a vector - we will start from a matrix initialized to have the desired dimensions.

```
A <- matrix(nrow = 3, ncol = 2)
```

```
A
```

```
     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA
[3,]   NA   NA
```

```
A <- matrix(0, nrow = 3, ncol = 2)
```

```
A <- matrix(1:(3*2), nrow = 3, ncol = 2)
```

Indexing Elements of/Subsetting a Matrix

```r
A
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```r
A[1, 2]
```

```
[1] 4
```

```r
A[1, ]
```

```
[1] 1 4
```

```r
A[, 1]
```

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

Avoiding Conversion to Vector

```r
A[1, , drop = FALSE]
```

```
     [,1] [,2]
[1,]    1    4
```

```r
A[, 1, drop = FALSE]
```

```
     [,1]
[1,]    1
[2,]    2
[3,]    3
```

Extracting the Dimensions of a Matrix

```r
dim(A)
```

```
[1] 3 2
```

```r
nrow(A)
```

```
[1] 3
```

```r
ncol(A)
```

```
[1] 2
```

Growing a Matrix

We can grow a matrix, but if we do so we need to add entire rows (`rbind`) or columns (`cbind`).

```
A <- rbind(A, c(0, 0))
```

```
A
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
[4,]    0    0
```

Growing a Matrix

We can grow a matrix, but if we do so we need to add entire rows (`rbind`) or columns (`cbind`).

```r
A <- cbind(A, c(0, 0, 0, 0), c(0, 0, 0, 0))
```

```r
A
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    0    0
[2,]    2    5    0    0
[3,]    3    6    0    0
[4,]    0    0    0    0
```

Indexing Diagonals of Matrices

```
A
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    0    0
[2,]    2    5    0    0
[3,]    3    6    0    0
[4,]    0    0    0    0
```

```
diag(A)
```

```
[1] 1 5 0 0
```

```
diag(A[1:2, ])
```

```
[1] 1 5
```

```
diag(A[-1, -ncol(A)])
```

```
[1] 2 6 0
```

Making Special Matrices

Another use of the `diag` function is to create a diagonal matrix.

```
diag(2)
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

```
diag(2, nrow = 2, ncol = 2)
```

```
     [,1] [,2]
[1,]    2    0
[2,]    0    2
```

```
diag(c(1, 2))
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    2
```

Subsetting a Matrix

```
A
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    0    0
[2,]    2    5    0    0
[3,]    3    6    0    0
[4,]    0    0    0    0
```

```
A <- A[1:3, 1:2]
```

Functions Summarizing Matrices

Matrices are just fancy vectors, and many of the functions we have discussed for summarizing vectors can be applied to a matrix. The function will just “flatten” the matrix into a vector first.

```r
max(A)
```

```
[1] 6
```

```r
range(A)
```

```
[1] 1 6
```

The functions `sum`, `mean`, `sd`, `var`, `min`, and `prod` work equivalently when applied to matrices.

Math for Matrices

Most of the functions that we described for vectors also apply to matrices.

```
A + 1
```

```
     [,1] [,2]
[1,]    2    5
[2,]    3    6
[3,]    4    7
```

This applies to `+`, `*`, `/`, `-`, `abs`, `round`, `sqrt`, `^`, and will work if we replace `1` above with another scalar or a matrix with the same dimensions as `A`.

What happens if we use mathematical functions like + with a matrix and a vector of a different size?

Operations on Matrices that Yield Logical Matrices

The following functions we introduced for vectors work for matrices too, and return matrices:

- Greater than `>`
- Greater than or equal to `>=`
- Less than `<`
- Less than or equal to `<=`
- Equal to `==`
- Not equal to `!=`
- `is.na`

Coercing a Matrix Back to a Vector

```r
c(A)
```

```
[1] 1 2 3 4 5 6
```

```r
as.numeric(A)
```

```
[1] 1 2 3 4 5 6
```

This is helpful for understanding what some functions do when applied to matrices.

Identifying Elements of A Matrix that Satisfy a Certain Condition

```r
which(A > 1)
```

```
[1] 2 3 4 5 6
```

```r
which(A > 1, arr.ind = TRUE)
```

```
     row col
[1,]   2   1
[2,]   3   1
[3,]   1   2
[4,]   2   2
[5,]   3   2
```

Subsetting a Matrix with Logical Vectors

As with vectors, we can also subset matrices using logical vectors.

```
A
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
A[A[, 1] >= 1, A[2, ] < 3]
```

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

Naming Rows/Columns of a Matrix

It can be very handy to give a matrix's rows and/or columns names, for easier use.

```
colnames(A) <- c("alpha", "beta")
row.names(A) <- c("a", "b", "c")
```

```
A
```

```
  alpha beta
a     1    4
b     2    5
c     3    6
```

Getting Rid of Row/Column Names

```
A
```

```
  alpha beta
a     1    4
b     2    5
c     3    6
```

```
colnames(A) <- NULL
row.names(A) <- NULL
```

```
A
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Special Summaries of Rows of a Matrix

- `rowSums`
- `rowMeans`

```r
rowMeans(A)
```

```
[1] 2.5 3.5 4.5
```

Special Summaries of Columns of a Matrix

- `colSums`
- `colMeans`

```r
colSums(A)
```

```
[1]  6 15
```

Summarizing Dimensions of a Matrix More Generally

The `apply` function allows us to apply functions along dimensions of a matrix.

```r
colSums(A)
```

```
[1]  6 15
```

```r
apply(A, 2, sum)
```

```
[1]  6 15
```

```r
apply(A, 2, function(x) {sum(x)})
```

```
[1]  6 15
```

Summarizing Dimensions of a Matrix More Generally

The `apply` function allows us to apply functions along dimensions of a matrix.

```r
rowSums(A)
```

```
[1] 5 7 9
```

```r
apply(A, 1, sum)
```

```
[1] 5 7 9
```

```r
apply(A, 1, function(x) {sum(x)})
```

```
[1] 5 7 9
```

What happens if we provide both array dimensions to `apply`, e.g. `apply(A, c(1, 2), sum)`?

Special Matrix Functions (Linear Algebra)

Matrix multiplication uses %*%.

```r
A%*%c(1, 2)
```

```
     [,1]
[1,]    9
[2,]   12
[3,]   15
```

The `t` function transposes a matrix.

```r
t(A)
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

The functions `crossprod` and `tcrossprod` can also be used to perform matrix multiplication, and sometimes yield speed benefits.

Special Matrix Functions (Linear Algebra)

For any matrix:

- `svd` returns the singular value decomposition

For square matrices:

- `solve` returns the inverse of a matrix
- `eigen` returns an eigendecomposition

For symmetric positive-definite square matrices:

- `chol` returns a cholesky decomposition