Vectors

Thanks to Dr.David Gerard

Learning Objectives

- Understanding vectors, the fundamental objects in R.
- This is mostly a long list of facts about vectors that you should be aware of.
- Chapter 5 of HOPR
- Chapter 3 from Advanced R

Vector Types

- Two types of vectors:
  - **Atomic**: All elements of same type.
  - **List** ("generic vectors"): Objects may be of different types.
  - (low-key third type) `NULL`: Absence of a vector.

Atomic Vectors

- Four basic types:
  - Logical: Either `TRUE` or `FALSE`
  - Integer:
    * Exactly an integer. Assign them by adding `L` behind it (for "long integer").
    * `-1L`, `0L`, `1L`, `2L`, `3L`, etc…
  - Double:
    * Decimal numbers.
    * `1`, `1.0`, `1.01`, etc…
    * `Inf`, `-Inf`, and `NaN` are also doubles.
  - Character:

    * Anything in quotes:
    * `"1"`, `"one"`, `"1 won one"`, etc…

- You create vectors with `c()` for "combine"

```r
x <- c(TRUE, TRUE, FALSE, TRUE) ## logical
x <- c(1L, 1L, 0L, 1L) ## integer
x <- c(1, 1, 0, 1) ## double
x <- c("1", "1", "0", "1") ## character
```

- There are no scalars in R.
- A "scalar" is just a vector length 1.

```r
is.vector(TRUE)
```

```
[1] TRUE
```

- Integers and doubles are together called "numerics"
- You can determine the type with `typeof()`.

```r
x <- c(TRUE, FALSE)
typeof(x)
```

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

```r
x <- c(0L, 1L)
typeof(x)
```

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

```r
x <- c(0, 1)
typeof(x)
```

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

```r
x <- c("0", "1")
typeof(x)
```

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

- The special values, `Inf`, `-Inf`, and `NaN` are doubles

```r
typeof(c(Inf, -Inf, NaN))
```

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

- Determine the length of a vector using `length()`

```r
length(x)
```

```
[1] 2
```

- Missing values are represented by `NA`.
- `NA` is technically a logical value.

```r
typeof(NA)
```

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

- This rarely matters because logicals get coerced to other types when needed.

```r
    typeof(c(1L, NA))
```

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

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

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

```r
    typeof(c("1", NA))
```

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

- But if you need missing values of other types, you can use

```
NA_integer_ ## integer NA
NA_real_ ## double NA
NA_character_ ## character NA
```

- Never use `==` when testing for missingness. It will return `NA` since it is always unknown if two unknowns are equal. Use `is.na()`.

```
x <- c(NA, 1)
x == NA
```

```
[1] NA NA
```

```
is.na(x)
```

```
[1]  TRUE FALSE
```

- You can check the type with `is.logical()`, `is.integer()`, `is.double()`, and `is.character()`.

```
is.logical(TRUE)
```

```
[1] TRUE
```

```
is.integer(1L)
```

```
[1] TRUE
```

```
is.double(1)
```

```
[1] TRUE
```

```
is.character("1")
```

```
[1] TRUE
```

- Attempting to combine vectors of different types coerces them to the same type. The order of preference is character $>$ double $>$ integer $>$ logical.

```
typeof(c(1L, TRUE))
```

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

```
typeof(c(1, 1L))
```

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

```
typeof(c("1", 1))
```

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

- **Exercise** (from Advanced R): Predict the output:

```
c(1, FALSE)
c("a", 1)
c(TRUE, 1L)
```

- **Exercise** (from Advanced R): Explain these results:

```
1 == "1"
```

```
[1] TRUE
```

```
-1 < FALSE
```

```
[1] TRUE
```

```
"One" < 2
```

```
[1] FALSE
```

Attributes

- Attributes are meta information applied to atomic vectors.
- Many common objects (like matrices, arrays, factors, date-times) are just atomic vectors with special attributes.
- You get and set attributes with `attr()`

```r
a <- 1:3
attr(a, "x") <- "abcdef" # sets x attribute of vector a to be "abcdef"
attr(a, "x") # retrieve the x attribute of vector a
```

```
[1] "abcdef"
```

- You can see all attributes of a vector with `attributes()`.

```r
attr(a, "y") <- 4:6

attributes(a)
```

```
$x
[1] "abcdef"

$y
[1] 4 5 6
```

- You can set many attributes at the same time with `structure()`.

```r
b <- structure(1:3,
               x = "abcdef",
               y = 4:6)
attributes(b)
```

```
$x
[1] "abcdef"

$y
[1] 4 5 6
```

- Attributes are name-value pairs, and all of these attributes are associated with an object. Below, the vector `c(1, 2, 3)` points to attributes `x` and `y` that each have their own values.

- Most attributes are typically lost by most operations.

```
attributes(a[[1]])
```

```
NULL
```

```
attributes(sum(a))
```

```
NULL
```

- **Exception**: Two attributes are not lost typically: **names** and **dim**.

Names

- Names are a character vector the same length as the atomic vector. Each name corresponds to a single element.
- You could set names using `attr()`, but you should not.

```
x <- 1:3
attr(x, "names") <- c("a", "b", "c")
attributes(x)
```

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

- Names are so special, that there are special ways to create them and view them

```
x <- c(a = 1, b = 2, c = 3)
names(x)
```

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

```
x <- 1:3
names(x) <- c("a", "b", "c")
names(x)
```

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

- The proper way to think about names is like this:

  But each name corresponds to a specific element, so Hadley does it like this:

- Names stay with single bracket subsetting (not double bracket subsetting)

```r
x
```

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

```r
names(x[1])
```

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

```r
names(x[1:2])
```

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

```r
names(x[[1]])
```

```
NULL
```

- Names can be used for subsetting (more in Chapter 4)

```r
x[["a"]]
```

```
[1] 1
```

- You can remove names with `unname()`.

```r
unname(x)
```

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

S3 Atomic Vectors

- One of the most important vector attributes is **class**, which underlies the S3 object system. Having a class attribute turns an object into an S3 object, which means it will behave differently from a regular vector when passed to a generic function.
- The class of an object will determine its behavior when you use that class in a **generic** function such as `print()` or `summary()`.
  - A generic function is a function that has different behavior based on the class of the input.
- You can create your own S3 classes (chapter 13).
- Here, we will talk about some S3 classes that come with R by default.
- You can determine the class of object with `class()`, and you can set the class to `NULL` by `unclass()`.
- Factors, Dates, and POSIXct (date-times)

FACTOR

- A **factor** is an integer vector with
  1. The `class` attribute `factor`, and
  2. A `levels` attribute describing the possible levels

```r
x <- factor(c("a", "b", "b", "a"))
x
```

```
[1] a b b a
Levels: a b
```

```r
typeof(x)
```

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

```r
class(x)
```

```
[1] "factor"
```

```r
attributes(x)
```

```
$levels
[1] "a" "b"

$class
[1] "factor"
```

- R also does some stuff under the hood for encoding factors (i.e. has a lot of methods specifically for factors).
- Factors are R's way of storing categorical variables, and are useful when a variable only has a certain number of possible values.
- Learn more about factors fromhere.

DATE

- A **Date** is a double vector with class attribute `Date`.

```r
today <- Sys.Date()
today
```

```
[1] "2024-02-13"
```

```r
typeof(today)
```

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

```r
attributes(today)
```

```
$class
[1] "Date"
```

```r
class(today)
```

```
[1] "Date"
```

- Let's look at the underlying double to today:

```r
unclass(today)
```

```
[1] 19766
```

- This is the number of days since January 1, 1970:

```r
unclass(as.Date("1970-01-01"))
```

```
[1] 0
```

POSIXct

- Date-time classes are called either `POSIXct` (Portable Operating System Interface in Unix, Calendar Time) or `POSIXlt` (Portable Operating System Interface in Unix, Local Time).
- `POSIXct` shows up more often. It is a double representing the number of seconds since the beginning of 1970.

```r
now <- Sys.time()
typeof(now)
```

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

```r
class(now)
```

```
[1] "POSIXct" "POSIXt"
```

```r
unclass(now)
```

```
[1] 1707867662
```

- `POSIXlt` is a named list of vectors with elements representing seconds, minutes, hours, days of the month, months, years, weekdays, etc…

```r
ltvec <- as.POSIXlt(x = c("1980-10-10 01:11:01",
                          "1970-01-11 10:15:22",
                                   "2010-05-30 20:01:18"))
typeof(ltvec)
```

```
[1] "list"
```

```r
unclass(ltvec)
```

```
$sec
[1]  1 22 18

$min
[1] 11 15  1

$hour
[1]  1 10 20

$mday
[1] 10 11 30

$mon
[1] 9 0 4

$year
[1]  80  70 110

$wday
[1] 5 0 0

$yday
[1] 283  10 149

$isdst
[1] 1 0 1

$zone
[1] "EDT" "EST" "EDT"

$gmtoff
[1] NA NA NA
```

```
attr(,"tzone")
[1] ""    "EST" "EDT"
attr(,"balanced")
[1] TRUE
```

- You mostly interact with these date-time objects through the `{lubridate}` package, but base R has their own interfaces (which I think are more difficult to use).
- Learn more about dates and date-times here.
- **Exercise** (From Advanced R): `table()` will take as input a vector or vectors and count how many observations have each value. What sort of object does `table()` return? What is its type? What attributes does it have? How does the dimensionality change as you tabulate more variables?

Creating Empty Vectors

- In many applications, you will want to create empty vectors or vectors filled with missing values.
- Create an empty vector with `vector()`.

```r
vector(mode = "character", length = 0)
```

```
character(0)
```

```r
vector(mode = "double", length = 0)
```

```
numeric(0)
```

```r
vector(mode = "integer", length = 0)
```

```
integer(0)
```

```r
vector(mode = "logical", length = 0)
```

```
logical(0)
```

- Shorthand for this is

```r
character()
```

```
character(0)
```

```r
double()
```

```
numeric(0)
```

```r
integer()
```

```
integer(0)
```

```r
logical()
```

```
logical(0)
```

- Empty vectors often show up in defaults that are returned when folks ask for something of length 0.
- E.g., in if you are simulating something, you might return a vector of length 0 if they ask for 0 elements.

```r
f <- function(n) {
  sout <- double(n)
  for (i in seq_len(n)) {
    sout[[i]] <- simcode(...) ## put simulation code here
  }
  return(sout)
}
```

- You often want to create an empty vector that you then fill in with values. I like to create this vector to be with missing values, so that I know I made a mistake if they are not all filled in.

```r
n <- 100
x <- rep(NA_character_, lenght.out = n)
x <- rep(NA_integer_, lenght.out = n)
x <- rep(NA_real_, lenght.out = n)
```

```r
x <- rep(NA, lenght.out = n)
```

- E.g. in a for-loop, you often fill in the elements of a vector. Let's suppose we are evaluating the performance of the mean in a simulation study.

```r
nsim <- 1000 ## number of simulations
nsamp <- 10 ## sample size
mvec <- rep(NA_real_, length.out = nsim)
true_mean <- 0
for (i in seq_len(nsim)) {
  mvec[[i]] <- mean(rnorm(n = nsamp, mean = true_mean))
}
mean((mvec - true_mean)^2) ## mean squared error
```

```
[1] 0.1090295
```

- If you are filling in the values of a matrix, you need to be able to create a matrix with missing values.

```r
n <- 100
p <- 3
matval <- matrix(NA_character_, nrow = p, ncol = n)
matval <- matrix(NA_real_, nrow = p, ncol = n)
matval <- matrix(NA_integer_, nrow = p, ncol = n)
matval <- matrix(NA, nrow = p, ncol = n)
```

Lists

- Lists are like vectors except each element can be of any type.
- You create lists with `list()`.

```r
lobj <- list(a = 1:3,
             log_val = TRUE,
             list(c = 10))
lobj
```

```
$a
[1] 1 2 3
```

```
$log_val
[1] TRUE

[[3]]
[[3]]$c
[1] 10
```

- You can view a list with `str()`.

```
str(lobj)
```

```
List of 3
 $ a      : int [1:3] 1 2 3
 $ log_val: logi TRUE
 $        :List of 1
  ..$ c: num 10
```

- `c()` will combine lists into a single list. If you use `c()` with a list and a vector, then it will first coerce the vector into a list where each element is a list.

```
l1 <- list(1:2,
           c("a", "b"))
l2 <- list(c(TRUE, FALSE))
c(l1, l2)
```

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

[[2]]
[1] "a" "b"

[[3]]
[1]  TRUE FALSE
```

```
c(l1, c("c", "d"))
```

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

```
[[2]]
[1] "a" "b"

[[3]]
[1] "c"

[[4]]
[1] "d"
```

```
as.list(c("c", "d")) ## this is what it does before combining
```

```
[[1]]
[1] "c"

[[2]]
[1] "d"
```

- `typeof()` will return `"list"` and `is.list()` tests for a list.

```
typeof(l1)
```

```
[1] "list"
```

```
is.list(l1)
```

```
[1] TRUE
```

- Use `unlist()` to remove the list structure.

```
l1
```

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

[[2]]
[1] "a" "b"
```

```
unlist(l1)
```

```
[1] "1" "2" "a" "b"
```

- The `dim` attribute can be applied to lists

```
lmat <- list(1:2,
             3:10,
             runif(4),
             c("Hello", "world"))
dim(lmat) <- c(2, 2)
lmat
```

```
     [,1]      [,2]
[1,] integer,2 numeric,4
[2,] integer,8 character,2
```

```
lmat[[1, 2]]
```

```
[1] 0.7564111 0.3691448 0.8793382 0.2168319
```

Data Frames

- Data Frames are lists where
    1. Each element is a vector.
    2. Each vector has the same length.

```
df <- data.frame(a = 4:6,
                 b = c("A", "B", "C"))
typeof(df)
```

```
[1] "list"
```

```
attributes(df)
```

```
$names
[1] "a" "b"

$class
[1] "data.frame"

$row.names
[1] 1 2 3
```

- Above, the "names" attribute are the columnames, and you can get them with `colnames()`

```r
colnames(df)
```

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

```r
names(df)
```

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

- The row.names are the row names, and you can obtain them with `row.names()` or `rownames()`.
  - `row.names()` are specifically for data frames, whereas `rownames()` was designed for extracting dimnames and was also altered to work with data frames.

```r
row.names(df)
```

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

```r
rownames(df)
```

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

- Those row names are automatically generated, but you can set them with `rownames()`.

```r
rownames(df) <- c("h", "i", "j")
df
```

```
  a b
h 4 A
i 5 B
j 6 C
```

- `tibbles`, from the package `{tibble}` are tidyverse data frames. The main differences are:

  1. Tibbles do not automatically coerce data (such as from strings to factors). Data frames used to do this in older versions of R.

```r
data.frame(x = c("a", "b", "c"),
           stringsAsFactors = FALSE) ## needed to be safe for older versions
```

```
  x
1 a
2 b
3 c
```

```r
tibble::tibble(x = c("a", "b", "c"))
```

```
# A tibble: 3 x 1
  x
  <chr>
1 a
2 b
3 c
```

  2. Tibbles do not change names if they happen to be non-syntactic (e.g. have spaces in them)

```r
data.frame(`hello world` = c(1, 2, 3))
```

```
  hello.world
1           1
2           2
3           3
```

```r
tibble::tibble(`hello world` = c(1, 2, 3))
```

```
# A tibble: 3 x 1
  `hello world`
          <dbl>
```

```
1             1
2             2
3             3
```

3. Tibbles will only recycle vectors of length 1.

```
data.frame(x = c(1, 2, 3, 4),
           y = c(1, 2))
```

```
  x y
1 1 1
2 2 2
3 3 1
4 4 2
```

```
tibble::tibble(x = c(1, 2, 3, 4),
               y = c(1, 2))
```

```
Error in `tibble::tibble()`:
! Tibble columns must have compatible sizes.
* Size 4: Existing data.
* Size 2: Column `y`.
i Only values of size one are recycled.
```

4. `{tibbles}` do not reduce to vectors when you subset one column. Folks disagree on whether this is good or bad.

```
df <- data.frame(`hello world` = c(1, 2, 3))
tib <- tibble::tibble(`hello world` = c(1, 2, 3))
attributes(df[, 1])
```

```
NULL
```

```
attributes(tib[, 1])
```

```
$names
[1] "hello world"

$row.names
[1] 1 2 3

$class
[1] "tbl_df"     "tbl"        "data.frame"
```

5. Data frames allow for row names, tibbles do not. Folks disagree on whether this is desirable (Hadley is extremely against it).
6. Tibbles print differently than data frames. Tibbles only print 10 rows and only the columns that will fit. But I actually prefer the data frame method better, because pretty doesn't matter when you are doing data analysis, and it's better to see all columns.

- **Exercise**: Based on our discussion of making zero-length vectors, create a data frame with zero rows and columns `a`, and `b`. Both should be double columns.

- **Exercise**: What does `data.frame()` do without any arguments?

- **Exercise**: Use the `row.names` argument of `data.frame()` to create a data frame with 100 rows and no columns.

NULL

- `NULL` is its own data type, that always has length 0.

```r
typeof(NULL)
```

```
[1] "NULL"
```

```r
length(NULL)
```

```
[1] 0
```

- `NULL` is used to represent an empty vector.

```r
c()
```

```
NULL
```

- `NULL` is often used as a default argument in a function for complicated arguments. The function operates one way unless a user specifies something for that argument. Look at `?ashr::ash.workhorse` for multiple examples.

```r
f <- function(x = NULL) {
  if (is.null(x)) {
    ## do something
  } else {
    ## do something else
  }
}
```

- E.g., let's create a function `wmean` that calculates a weighted mean if weights are provided, an the sample mean otherwise.

```r
wmean <- function(x, w = NULL) {
  if (is.null(w)) {
    w <- rep(1 / length(x), length.out = length(x))
  } else {
    w <- w / sum(w)
  }
  return(sum(x * w))
}

x <- c(1, 2, 3)
wmean(x)
```

```
[1] 2
```

```r
wmean(x, w = c(5, 1, 1))
```

```
[1] 1.428571
```

- There are two alternative strategies to this. First, use `missingArg()` to test if an argument is missing.

```r
wmean <- function(x, w) {
  if (missingArg(w)) {
    w <- rep(1 / length(x), length.out = length(x))
  } else {
    w <- w / sum(w)
  }
  return(sum(x * w))
}

x <- c(1, 2, 3)
wmean(x)
```

```
[1] 2
```

```r
wmean(x, w = c(5, 1, 1))
```

```
[1] 1.428571
```

```
This works because of lazy evaluation (which we will learn about later).\
I don't like this because it is confusing to the user, who thinks `w` is a required argument
```

- Second, you can include more complicated defaults.

```r
wmean <- function(x, w = rep(1, length(x))) {
  w <- w / sum(w)
  return(sum(x * w))
}

x <- c(1, 2, 3)
wmean(x)
```

```
[1] 2
```

```r
wmean(x, w = c(5, 1, 1))
```

```
[1] 1.428571
```

```
I don't like this because default arguments are evaluated inside the function, but user-provi
```

- `NULL` is one of the ways R handle's missingness. The others are `NA` and `NaN`.
    - `NULL`: An empty object. Can be thought of as a zero-length vector.
    - `NA`: A missing value. Can be used as an element of a vector.
    - `NaN`: Undefined numeric values, such as the output of `0/0`.

New Functions

- `typeof()`: Determine the type of an object (character, double, integer, or logical).
- `attr()`: Get or set an attribute.
- `attributes()`: View all attributes.
- `structure()`: Create an object with many attributes.
- `names()`: Get or set names attributes.
- `unname()`: Remove the names attribute.
- `dim()`: Get or set dim attributes.
- `class()`: Get or set class attributes.
- `unclass()`: Remove the class attribute.