advanced-r

Jane Doe

7/9/2022

Table of contents

Introduction

This book contains all of my notes and exercise work-throughs from Hadley Wickham's Advanced R, 2nd edition.

Source code is organized at the chapter-section level. In each section, notes appear first, followed by the exercises. Exercise question text are written in *italics*.

This book was rendered using Quarto.

1 Advanced R Chapter 2 - Names and Values

Book Link

1.0.1 Notes

- A value does not have a name; a name has a value.
- That is, a name gets bound to a value as a reference to that value.
- Assigning a second name to that object does not change the object or copy it. It simply assigns a new name as reference to that object.
- Names have to be pretty. See `?make.names` for rules governing syntactically valid names.

1.0.2 Exercises

1. Explain the relationship between `a`, `b`, `c`, and `d` in the following code:

```
a <- 1:10
b <- a
c <- b
d <- 1:10
```

`a`, `b`, and `c` all occupy the same space in memory. Thus, they are separate references, each of which have been assigned to the same object.

```
library(lobstr)
obj_addr(a) == obj_addr(b) & obj_addr(b) == obj_addr(c)
```

```
[1] TRUE
```

```
print(obj_addr(a))
```

```
[1] "0x7fc62de08fb0"
```

However, `d` does not occupy the same memory space. Thus, `d` is an is a separate object from that of `a`, `b`, and `c`, with a unique reference. Both of these objects are the vector of integers 1 through 10.

```r
obj_addr(d) == obj_addr(a)
```

```
[1] FALSE
```

```r
print(obj_addr(d))
```

```
[1] "0x7fc62df084f8"
```

2. The following code accesses the mean function in multiple ways. Do they all point to the same underlying function object?

All accessors to the `mean()` function point to the same object in memory.

```r
objs <- list(mean, base::mean, evalq(mean), match.fun("mean"))
obj_addrs(objs)
```

```
[1] "0x7fc62e1f0400" "0x7fc62e1f0400" "0x7fc62e1f0400" "0x7fc62e1f0400"
```

3. By default, base R data import functions, like `read.csv()`, will automatically convert non-syntactic names to syntactic ones. Why might this be problematic? What option allows you to suppress this behavior?

Column names often represent data, so renaming with `make.names` changes underlying data. You can suppress this with `check.names = FALSE`.

4. What rules does `make.names()` use to convert non-syntactic names into syntactic ones?

From the docs, *"A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number."* Letters are defined by locale, but only ASCII digits are used. Invalid characters are translated to ".". Missing is translated to "NA". And reserved words have a "." appended to them. Then, values are de-duplicated using `make.unique()`.

5. I slightly simplified the rules that govern syntactic names. Why is `.123e1` not a syntactic name?

Syntactic names may start with a letter, or a dot not followed by a number. `.123e1` starts with `.1`, so it is not a syntactically valid name.

1.1 2.3: Copy-on-modify

1.1.1 Notes

Consider the following two variables

```r
x <- c(1, 2, 3)
y <- x
```

Note from previously that `x` and `y` are different references to the same object.

```r
obj_addr(x) == obj_addr(y)
```

```
[1] TRUE
```

This object is located at the following address:

```r
obj_addr(y)
```

```
[1] "0x7fc62d219778"
```

Modifying the object assigned to `y` results in the creation of a new object.

```r
y[[3]] <- 4
obj_addr(y)
```

```
[1] "0x7fc61d6d7468"
```

We see that this is different than the original object's address

```r
obj_addr(x) == obj_addr(y)
```

```
[1] FALSE
```

This behavior is called ***copy-on-modify***; i.e., R objects are immutable – any changes results in the creation of a new object in memory.

1.1.1.1 tracemem()

`base::tracemem()` will track an object's location in memory.

```
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
```

```
<0x7fc61cfbdee8>
```

In the example below, a second name, `y` was assigned to an object, which already had an assigned name `x`. So when `x` or `y` is modified, copy-on-modify takes place. This will trigger `tracemem()` to print a memory change.

```
y <- x
y[[4]] <- 4L
```

```
tracemem[0x7fc61cfbdee8 -> 0x7fc60d049208]: eval eval eval_with_user_handlers withVisible wi
```

`base::untracemem()` is the opposite of `base::tracemem()`

```
untracemem(x)
```

1.1.1.2 Lists

Lists do not store values. Instead, they store references to them.

When you modify elements of a list, you can view how this effects copy-on-modify behavior using `lobstr::ref()`.

Here, we assign a new reference to a list object. We then modify one of the elements in the list, triggering copy-on-modify. With `lobstr::ref()`, we see that a new object was created for the list itself and the modified object. The other elements of the list remain the same object.

In the output below, the first and fifth references are the lists themselves. The sixth references points to the new object, 4.

```
l1 <- list(1, 2, 3)
l2 <- l1
l1[[3]] <- 4
ref(l1, l2)
```

```
 [1:0x7fc60c262d98] <list>
[2:0x7fc61caa9a50] <dbl>
[3:0x7fc61caa9a18] <dbl>
[4:0x7fc61caa9900] <dbl>

 [5:0x7fc60c1f1de8] <list>
[2:0x7fc61caa9a50]
[3:0x7fc61caa9a18]
[6:0x7fc61caa99e0] <dbl>
```

1.1.1.3 Data frames

Since data frames are a list of columns, and those columns are vectors, modifying a column only results in that column being copied:

```
d1 <- data.frame(a = c(1, 2, 3), b = c(4, 5, 6))
tracemem(d1)
```

```
[1] "<0x7fc62eb058c8>"
```

Here, `tracemem()` shows us that the new column was copied to a new object in memory.

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
```

```
tracemem[0x7fc62eb058c8 -> 0x7fc60d0a6508]: eval eval eval_with_user_handlers withVisible wi
tracemem[0x7fc60d0a6508 -> 0x7fc60d0a6448]: [<-.data.frame [<- eval eval eval_with_user_hand
```

And with `lobstr::ref()`, we confirm that both the data.frame object and the second column were copied.

```
ref(d1, d2)
```

```
tracemem[0x7fc62eb058c8 -> 0x7fc60d194788]: FUN lapply ref eval eval eval_with_user_handlers
tracemem[0x7fc60d0a6448 -> 0x7fc60d208908]: FUN lapply ref eval eval eval_with_user_handlers
```

```
 [1:0x7fc62eb058c8] <df[,2]>
a = [2:0x7fc60c4e3648] <dbl>
b = [3:0x7fc60c4e35f8] <dbl>
```

```
 [4:0x7fc60d0a6448] <df[,2]>
a = [2:0x7fc60c4e3648]
b = [5:0x7fc60c4f3bb8] <dbl>
```

Since data.frames are built column-wise, modifying a row results in copying every column.

```r
d3 <- d1
d1[1, ] <- d1[1, ] * 2
```

```
tracemem[0x7fc62eb058c8 -> 0x7fc60d38bb48]: eval eval eval_with_user_handlers withVisible wi
tracemem[0x7fc60d38bb48 -> 0x7fc60d38ba08]: [<-.data.frame [<- eval eval eval_with_user_handl
```

```r
untracemem(d1)
ref(d1, d3)
```

```
tracemem[0x7fc62eb058c8 -> 0x7fc60d31a308]: FUN lapply ref eval eval eval_with_user_handlers
```

```
 [1:0x7fc60d38ba08] <df[,2]>
a = [2:0x7fc60c9bd8c8] <dbl>
b = [3:0x7fc60c9bd878] <dbl>

 [4:0x7fc62eb058c8] <df[,2]>
a = [5:0x7fc60c4e3648] <dbl>
b = [6:0x7fc60c4e35f8] <dbl>
```

1.1.1.4 Character vectors

R uses a global string pool in each session. This means that each element of a character vector points to a string in the globally unique pool. The references can be viewed in `lobstr::ref()` by setting `character` to `TRUE`.

```r
x <- letters[1:3]
ref(x, character = TRUE)
```

```
 [1:0x7fc62e0b3258] <chr>
[2:0x7fc63ca7eae8] <string: "a">
[3:0x7fc63c43b240] <string: "b">
[4:0x7fc63c00e0c0] <string: "c">
```

1.1.2 Exercises

1. Why is `tracemem(1:10)` not useful?

`1:10` is a sequence no name assigned to it, therefore will not be traceable after this initial call.

2. Explain why `tracemem()` shows two copies when you run this code. Hint: carefully look at the difference between this code and the code shown earlier in the section.

```
x <- c(1L, 2L, 3L)
tracemem(x)
```

```
[1] "<0x7fc60c208488>"
```

```
x[[3]] <- 4
```

```
tracemem[0x7fc60c208488 -> 0x7fc60c2d2988]: eval eval eval_with_user_handlers withVisible wi
tracemem[0x7fc60c2d2988 -> 0x7fc60d02b368]: eval eval eval_with_user_handlers withVisible wi
```

```
untracemem(x)
```

In this example, the original object `x` is an integer vector. When the third element of the vector is modified to 4, a double, the vector is first modified by being converted to a double vector. Then it is modified again when the third element is modified. This results in two copies-on-modify, reflected in the `tracemem()` output.

1.2 2.4: Object size

1.2.1 Notes

`lobstr::obj_size()` shows you the size of an object in memory.

```
obj_size(iris)
```

```
7,200 B
```

```
obj_size(c(1, 2, 3))
```

```
80 B
```

1.2.1.1 Lists

Because lists store references to objects, a repeated list is smaller than one would imagine. This is because the reference occupies less space in memory than the object itself.

```r
l1 <- list(rep(100, 100000))
obj_size(l1)
```

```
800,104 B
```

```r
l2 <- list(l1, l1, l1)
obj_size(l2)
```

```
800,184 B
```

A list repeated three times is 80 bytes larger than the original list, which was of size 800,104 bytes. 80 bytes is the size of a list of three empty objects:

```r
obj_size(list(NULL, NULL, NULL))
```

```
80 B
```

1.2.1.2 Character vectors

Similarly, since character vectors are references to the global string pool, a repeated character vector does not increase size dramatically:

```r
s <- "ask me about my sentence"
obj_size(s)
```

```
136 B
```

```r
obj_size(rep(s, 10))
```

```
256 B
```

1.2.1.3 Alternative representation

Starting in R 3.5.0, alternative representation allowed R to represent certain vectors compactly. This is most commonly observed when using `:` to create a sequence.

Because of ALTREP, every sequence in R has the same size, regardless of the sequence length:

```r
obj_size(1:2)
```

```
680 B
```

```r
obj_size(1:10)
```

```
680 B
```

```r
obj_size(1:1e9)
```

```
680 B
```

1.2.2 Exercises

1.In the following example, why are object.size(y) and obj_size(y) so radically different?

```r
y <- rep(list(runif(1e4)), 100)

object.size(y)
```

```
8005648 bytes
```

```r
obj_size(y)
```

```
80,896 B
```

Per the documentation, `base::object.size()` measures the size of each object, and does not account for shred objects within lists.

2. Take the following list. Why is its size somewhat misleading?

```r
funs <- list(mean, sd, var)
obj_size(funs)
```

```
17,608 B
```

These objects come shipped with base R, so they are always available. It does not represent additional memory allocated to these objects.

1.3 2.5: Modify-in-place

1.3.1 Notes

There are two exceptions to copy-on-modify:

- Objects with a single binding
- Environments

In these exceptions, R executes a modify in place optimization.

1.3.1.1 Objects with a single binding

If an object has only one name assigned to it, R will modify in place.

Before:

```r
a <- c(1, 2, 3)
ref(a)
```

```
[1:0x7fc61d6d80e8] <dbl>
```

After:

```r
a[[3]] <- 4
ref(a)
```

```
[1:0x7fc60d030af8] <dbl>
```

Note that this optimization does *not* apply when modifying a vector's length. Here, `z` is assigned to a new object upon "adding" a fourth element to the vector `z`.

Before:

```r
z <- letters[1:3]
obj_addr(z)
```

```
[1] "0x7fc60d09cb58"
```

After:

```r
z[[4]] <- "d"
obj_addr(z)
```

```
[1] "0x7fc60d411468"
```

There are two complications in R's behavior that limit execution of the modify-in-place optimization:

1. R can only count if an object has 0, 1, or many references. That means that if an object has one of two bindings removed, the binding count will remain at many and modify-in-place will not apply.
2. Most functions make a reference of the object to be modified, so copy-on-modify would apply. The exception are "primitive" C functions, found mostly in the base package.

For example, modifying a data frame results in additional references being made, whereas modifying a list uses internal C code that does not create new references.

It's best to confirm copy behavior using `tracemem()`.

First, on modifying a data frame:

```r
x <- as.data.frame(matrix(runif(1e3), ncol = 4))
tracemem(x)
```

```
[1] "<0x7fc60c4f4ce8>"
```

```r
for (i in seq_along(x)) {
  x[[i]] <- x[[i]] * 5
}
```

```
tracemem[0x7fc60c4f4ce8 -> 0x7fc60c8badd8]: eval eval eval_with_user_handlers withVisible wi
tracemem[0x7fc60c8badd8 -> 0x7fc60c8bace8]: [[<-.data.frame [[<- eval eval eval_with_user_han
tracemem[0x7fc60c8bace8 -> 0x7fc60c8bac48]: eval eval eval_with_user_handlers withVisible wi
tracemem[0x7fc60c8bac48 -> 0x7fc60c8ba978]: [[<-.data.frame [[<- eval eval eval_with_user_han
tracemem[0x7fc60c8ba978 -> 0x7fc60c8ba838]: eval eval eval_with_user_handlers withVisible wi
tracemem[0x7fc60c8ba838 -> 0x7fc60c8ba748]: [[<-.data.frame [[<- eval eval eval_with_user_han
tracemem[0x7fc60c8ba748 -> 0x7fc60c8ba6a8]: eval eval eval_with_user_handlers withVisible wi
tracemem[0x7fc60c8ba6a8 -> 0x7fc60c8ba5b8]: [[<-.data.frame [[<- eval eval eval_with_user_han
```

```r
untracemem(x)
```

Then a list:

```r
l <- as.list(x)
tracemem(l)
```

```
[1] "<0x7fc60c9bcb88>"
```

```r
for (i in seq_along(l)) {
  l[[i]] <- l[[i]] * 5
}
```

```
tracemem[0x7fc60c9bcb88 -> 0x7fc60c9c5d78]: eval eval eval_with_user_handlers withVisible wi
```

```r
untracemem(l)
```

1.3.1.2 Environments

Environments are always modified-in-place, since existing bindings in that environment continue to have the same reference.

1.3.2 Exercises

1. *Explain why the following code doesn't create a circular list.*

```r
x <- list()
x[[1]] <- x
```

In the first line of code, an empty list is created, and the name `x` assigned to it.

The next line of code modifies the length of `x`, so copy-on-modify takes place to create a new list of length one, with the first element being `x`, the original empty list.

This is confirmed by observing the matching memory addresses:

Original list:

```r
x <- list()
ref(x)
```

```
[1:0x7fc62e5e2878] <list>
```

Modified list:

```r
x[[1]] <- x
ref(x)
```

```
[1:0x7fc62e0c5408] <list>
 [2:0x7fc62e5e2878] <list>
```

2. Wrap the two methods for subtracting medians into two functions, then use the 'bench' package to carefully compare their speeds. How does performance change as the number of columns increase?

Note: Instead of multiplying columns by 5 to demonstrate the exception to modify-in-place, the book subtracted medians. I'll continue to multiply by 5 below.

With a 250 x 4 data set, the differences in both speed is about 6-8x.

```r
library(bench)
mult_five_seq <- function(x) {
  for (i in seq_along(x)) {
    x[[i]] <- x[[i]] * 5
  }
}

df <- as.data.frame(matrix(runif(1e4), ncol = 4))
l <- as.list(df)

bm <- mark(df = mult_five_seq(df), l = mult_five_seq(l))
knitr::kable(bm[, 1:5])
```

expression	min	median	itr/sec	mem_alloc
df	46.21us	49us	19107.13	99.9KB
l	3.79us	5us	186723.91	78.3KB

With a 250 x 400 data set, the differences in both speed and memory allocation are much more pronounced. Here, 70x faster and a quarter of the memory used.

```r
df <- as.data.frame(matrix(runif(1e5), ncol = 400))
l <- as.list(df)

bm <- mark(df = mult_five_seq(df), l = mult_five_seq(l))
knitr::kable(bm[, 1:5])
```

expression	min	median	itr/sec	mem_alloc
df	7.85ms	8ms	124.9668	3.26MB
l	94.79us	101us	9445.2901	803.17KB

3. What happens if you attempt to use `tracemem()` on an environment? Modify-in-place always applies to environments, since existing bindings keep their references.

1.4 2.6: Unbinding the garbage collector

1.4.1 Notes

R uses a **garbage collector** to automatically free up unused memory when it is needed for a new object. You can force a garbage collection with `gc()`, but there's never any need to call it.

An example of unused memory are objects that no longer have a reference.

```r
x <- 1:3
x <- 2:4
rm(x)
```

Two objects were created above as a result of the original object creation and the copy-on-modify behavior. Then, the name `x` was removed, but the objects remained. R's garbage collector will remove these objects when necessary. You can have the collector print a message every time it runs with `gcinfo(TRUE)`.

2 Advanced R Chapter 3 - Vectors

Book Link

- Vectors, an important R data type, have two types: *atomic vectors* and *lists* (generic vectors)
- Vectors also have metadata in the form of *attributes*

2.0.1 Notes

- Four primary types of atomic vectors
  - logical
  - character
  - double
  - integer
    - Together, double and integer are numeric vectors
- Two rare types:
  - complex
  - raw

2.0.1.1 Scalars

Scalars, aka individual values, are created in special ways for each of the four primary types:

Type	Value
Logical	`TRUE` or `FALSE` (or `T` or `F`)
Character	surrounded by `"` or `'`. see `?Quotes` for escape characters
Double	decimal (`0.123`), scientific (`1.23e3`), or hexadecimal form
Integer	similar to doubles, ending with `L` (`123L`)

- There are three special values unique to doubles: `Inf`, `-Inf`, and `NaN` (not a number)

2.0.1.2 c()

- `c()`, short for combine, is used to create longer vectors
- Determine the vector type with `typeof()`
- If the inputs to `c()` are other atomic vectors, R will flatten them into one atomic vector.

```
x1 <- c(1, 2)
x2 <- c(3, 4)
c(x1, x2)
```

```
[1] 1 2 3 4
```

2.0.1.3 Missing values

- Missing values are represented with `NA`
  - Each primary type has its own missing value (R usually converts to correct type)

Type	Missing Value
Logical	`NA`
Character	`NA_character`
Double	`NA_real_`
Integer	`NA_integer_`

- Most computations involving missing values will return a missing value, with a few exceptions:

The 0 power identity

```
NA ^ 0
```

```
[1] 1
```

Boolean logic (or TRUE):

```
NA | TRUE
```

```
[1] TRUE
```

Boolean logic (and FALSE):

```r
NA & FALSE
```

```
[1] FALSE
```

- Use `is.na()` to check for missingness in vectors

```r
x <- c(NA, 2, 3, NA)
is.na(x)
```

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

2.0.1.4 Testing

The primary types can be checked with the appropriate `is.*()` function:

Type	`is.*()` function
Logical	`is.logical()`
Character	`is.character()`
Double	`is.double()`
Integer	`is.integer()`

2.0.1.5 Coercion

Coercion to a different type often happens automatically as a result of a computation. To deliberately coerce, use the appropriate `as.*()` function:

Type	`as.*()` function
Logical	`as.logical()`
Character	`as.character()`
Double	`as.double()`
Integer	`as.integer()`

Failed coercion generates a warning and returns `NA` for that value.

```r
as.integer(c("1", "2.5", "bike", "7"))
```

```
Warning: NAs introduced by coercion
```

```
[1]  1  2 NA  7
```

Per `?c`, the hierarchy of types when coercing is NULL < raw < logical < integer < double < complex < character < list < expression

2.0.2 Exercises

1. How do you create raw and complex scalars? (See `?raw` and `?complex`.)

Raw vectors are created with `raw()`, specifying the single `length` argument. Raw vectors also have a specific `is.raw()` for checking and `as.raw()` for coercion.

```r
r <- raw(2)
r
```

```
[1] 00 00
```

Complex vectors are created with `complex()`, specifying either the length (via the `length.out` argument) or both the real and imaginary parts as numeric vectors.

2. Test your knowledge of the vector coercion rules by predicting the output of the following uses of `c()`

Per the type coercion hierarchy, `c(1, FALSE)` will be converted to a double, `c(1, 0)`. 1's are coerced to `TRUE` (and 0 to `FALSE`), and vice versa

```r
x <- c(1, FALSE)
cat(x, "\n", typeof(x))
```

```
1 0 
 double
```

`c("a", 1)`, will be coerced to `c("a", "1")` as the character type is about double in the type coversion hierarchy

```r
x <- c("a", 1)
cat(x, "\n", typeof(x))
```

```
a 1 
 character
```

`c(TRUE, 1L)`, will b coerced to `c(1L, 1L)` per the type coercion hierarchy

```r
x <- c(TRUE, 1L)
cat(x, "\n", typeof(x))
```

```
1 1
 integer
```

3.1. Why is `1 == "1"` true?

Per the documentation for `==`, atomic vectors that are of different types are coerced prior to evaluation. So, the left hand of the equality, `1`, will be coerced to a character vector prior to evaluation. Once coerced, the call is `"1" == "1"`, which is clearly true.

3.2. Why is `-1 < FALSE` true?

`<`, like `==`, also coerces types prior to evaluation. So after coercion, the call here becomes `-1 < 0`, which is true.

3.3. Why is `"one" < 2` false?

Per the documentation for `<`, string comparison is done at the locale level. The locale in use can be viewed with `Sys.getlocale()` and similarly set with `Sys.setlocale()`. Because numbers come before letters in this sequence, the coerced inequality, `"one" < "2"` is evaluated as `FALSE`.

4. Why is the default missing value, `NA`, a logical vector? What's special about logical vectors? (Hint: think about `c(FALSE, NA_character_)`.)

Logical vectors are lowest on the type hierarchy. When they are combined with another primary type, logicals will always be coerced into the other type.

5. Precisely what do `is.atomic()`, `is.numeric()`, and `is.vector()` test for?

- `is.atomic()` checks if an object is of type "logical", "integer", "numeric", "complex", "character" or "raw"
- `is.numeric()` checks if an object is a double or integer vector and *not* a factor.
- `is.vector()` is a generalized `is.*()` function for all vectors. The `mode` argument can be specified to check for a specific type, including lists. It can also be left as "any", the default, to check is the object is a vector. `mode` can also be specified as "numeric", running the same check as `is.numeric()`.

2.1 3.3: Attributes

2.1.1 Notes

2.1.1.1 Getting and setting

- Attributes are name-value pairs that attach metadata to an R object
- Individual attributes can be get and set with `attr()`
- Attributes are retrieved en masse with `attributes()` and set with `structure()`
- Most attributes, other than **dim** and **names** are lost by most operations
  - To define and preserve attributes, create an S3 class, discussed in Chapter 13

2.1.1.2 Names

- Names can be set in three ways:

```r
# When creating it:
x <- c(a = 1, b = 2, c = 3)

# By assigning a character vector to names()
x <- 1:3
names(x) <- c("a", "b", "c")

# Inline, with setNames():
x <- setNames(1:3, c("a", "b", "c"))
```

- Names should be unique and non-missing, though this is not enforced in R

2.1.1.3 Dimensions

- Adding the **dim** attribute allows a vector to behave like a 2-d **matrix** or multi-dimensional **array**
  - You can also create matrices and arrays with `matrix()` and `array()`
- If a vector has no `dim` attribute set, that is equivalent to a vector with `NULL` dimensions
- Many functions for working with vectors have generalizations for working with matrices and arrays

2.1.2 Exercises

1. How is `setNames()` implemented? How is `unname()` implemented? Read the source code.

`setNames()` assigns names to the object by calling `names()<-`

```r
setNames
```

```
function (object = nm, nm)
{
    names(object) <- nm
    object
}
<bytecode: 0x7fb3bb9e8d18>
<environment: namespace:stats>
```

`unname()` sets names to `NULL` using `names()<-`. If the object is a data frame, or `force` is set to `TRUE`, the dimension names are set to `NULL` using `dimnames()<-`

```r
unname
```

```
function (obj, force = FALSE)
{
    if (!is.null(names(obj)))
        names(obj) <- NULL
    if (!is.null(dimnames(obj)) && (force || !is.data.frame(obj)))
        dimnames(obj) <- NULL
    obj
}
<bytecode: 0x7fb3d8ec7898>
<environment: namespace:base>
```

2.1 What does dim() return when applied to a 1-dimensional vector?

The dimensions of a 1-d vector are `NULL`, so `dim()` returns `NULL`.

```r
x <- c(1, 2, 3)
dim(x)
```

```
NULL
```

2.2 When might you use NROW() or NCOL()?

The difference between `NROW()` and `nrow()` (and `NCOL()` and `ncol()`) is that the capitalized forms return values for one dimensional vectors. Both functions return the same value for matrices, arrays, and data frames.

For one dimensional objects: - `NROW()` returns the length of the vector - `NCOL()` returns `1L` >

3. How would you describe the following three objects? What makes them different from `1:5`?

```
x1 <- array(1:5, c(1, 1, 5))
x2 <- array(1:5, c(1, 5, 1))
x3 <- array(1:5, c(5, 1, 1))
```

`x1`, `x2`, and `x3` are all one dimensional arrays in a 3 dimensional space. Each has a numeric vector of length 3 as the `dim` attribute. `1:5`, being a 1 dimensional vector, does not have a `dim` attribute.

```
dim(1:5)
```

```
NULL
```

4. An early draft used this code to illustrate `structure()`:

```
structure(1:5, comment = "my attribute")
```

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

But when you print that object you don't see the comment attribute. Why? Is the attribute missing, or is there something else special about it? (Hint: try using help.)

Checking the documentation for the default method for `print()`, (`?print.default`), we see that attributes are printed depending on an object's class(es).

`1:5` is stored as an "integer"

```
class(1:5)
```

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

Since there is no `print` method defined for “integer”, `print.default()` will be used and no attributes printed.

If we wanted the attribute `comment` to be printed with the output, we would have to create an S3 class with and attribute `comment` and define a `print` method for that class that includes the `comment` attribute.

2.2 3.4: S3 atomic vectors

2.2.1 Notes

Discussed more in Chapter 13, S3 objects have a `class` attribute, which means it will have special behavior for generic functions.

There are four important S3 vectors in base R:

- Categorical data in **factor** vectors, an integer
- Dates in **Date** vectors, a double
- Date-times in **POSIXct** vectors, a double
- Durations in **difftime** vectors, a double

2.2.1.1 Factors

Factors are useful when looking at categorical data. They sit on top of integers with two attributes:

- `class`: “factor”
- `levels`: defines factor’s categories

Ordered factors are just like factors, except that the order of the factors is meaningful

2.2.1.2 Dates

Dates are built on double vectors with a `class` attribute “Date”.

The value of the double represents the number of days since 1970-01-01.

```r
d <- as.Date("1971-01-01")
unclass(d)
```

```
[1] 365
```

2.2.1.3 Date-times

R stores date-time data in two ways, POSIXct, and POSIXlt. POSIX stands for Portable Operating System Interface, ct for calendar time, and lt for local time. POSIXct is the simplest usage.

POSIXct variables are built on top of double variables, with two attributes:

- `class`: "POSIXct"
- `tzone`: "UTC", "GMT", " " for local, or a timezone name

The `tzone` attributes controls the timezone, and can be modified using `attr()` or `structure()`. See this Wikipedia page for a list of the timezone names and `?timezones` for where to locate the tz database on your system.

2.2.1.4 Durations

Durations, representing the time between two dates or date-times, are stored in **difftimes**.

Difftimes are built on doubles with a `units` attribute that determines how the duration should be calculated.

2.2.2 Exercises

1. What sort of object does `table()` return? What is its type? What attributes does it have? How does the dimensional change as you tabulate more variables?

From viewing its structure with `str()`, we see that `t` is a built on an integer, appearing to be a 1-dimensional array. It has an additional attribute `dimnames` containing the factor levels.

```r
x <- c("cat", "dog", "dog")
f <- factor(x, levels = c("cat", "dog", "horse"))
t <- table(f)
str(t)
```

```
 'table' int [1:3(1d)] 1 2 0
 - attr(*, "dimnames")=List of 1
  ..$ f: chr [1:3] "cat" "dog" "horse"
```

Looking at `dim(t)` we confirm that this is a 1 dimensional array.

```r
dim(t)
```

```
[1] 3
```

`attributes()` also provides a clean look at `t`

```r
attributes(t)
```

```
$dim
[1] 3

$dimnames
$dimnames$f
[1] "cat"   "dog"   "horse"


$class
[1] "table"
```

For each variable added to the tabulation, the dimensionality increases by 1. Here with two variables, we note that `t2` is a two dimensional array.

```r
y <- c("female", "male", "female")
g <- factor(y)
t2 <- table(f, g)
attributes(t2)
```

```
$dim
[1] 3 2

$dimnames
$dimnames$f
[1] "cat"   "dog"   "horse"

$dimnames$g
[1] "female" "male"


$class
[1] "table"
```

And `t3`, with three tabulated variables, is a three dimensional array.

```r
z <- c("black", "black", "brown")
h <- factor(z)
t3 <- table(f, g, h)
attributes(t3)
```

```
$dim
[1] 3 2 2

$dimnames
$dimnames$f
[1] "cat"   "dog"   "horse"

$dimnames$g
[1] "female" "male"

$dimnames$h
[1] "black" "brown"


$class
[1] "table"
```

2. What happens to a factor when you modify its levels?

Using `tracemem()`, we see that this is considered a modification, so copy-on-modify takes place.

```r
f1 <- factor(letters)
tracemem(f1)
```

```
[1] "<0x7fb3db759ee8>"
```

```r
levels(f1) <- rev(levels(f1))
```

```
tracemem[0x7fb3db759ee8 -> 0x7fb3db75aca8]: eval eval eval_with_user_handlers withVisible wi
```

3. What does this code do? How do `f2` and `f3` differ from `f1`?

```r
f2 <- rev(factor(letters))
f3 <- factor(letters, levels = rev(letters))
```

The key difference here is the order of the levels in each output. Since `rev()` reverses the vector's values, not the values the `levels` attributes the levels for `f2`, are in alphabetical order:

```r
levels(f2)[1:5]
```

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

For `f3`, the levels are in reverse alphabetical order, as `rev()` was called on the levels themselves when it was created.

```r
levels(f3)[1:5]
```

```
[1] "z" "y" "x" "w" "v"
```

2.3 3.5: Lists

2.3.1 Notes

2.3.1.1 Creating

Lists can be created using `list()`.

- Lists contain references to other objects
- A list's those objects can be of any type, including other lists
- Because a list can contain another list, lists are sometimes called **recursive** vectors
- `c()` combines multiple lists into one. If some of the inputs to `c()` are atomic vectors and others lists, R will coerce the vectors to lists prior to combining

2.3.1.2 Testing and coercion

- `typeof()` returns "list" for lists
- Check if a list with `is.list()` and coerce to a list with `as.list()`
- You can also coerce a list to an atomic vector with `unlist()`
    - From the book, The rules for the type resulting from `unlist()` are *"complex, not well documented, and not always equivalent to what you'd get with `c()`"*

2.3.1.3 Matrices and arrays

Similarly to atomic vectors, you *can* add a `dim` attribute to lists to create list-matrices or list-arrays, if you want to.

2.3.2 Exercises

1. List all the ways that a list differs from an atomic vector.

- Because a list contains references, the list's elements can be of any type
- A list stores references to other objects in memory, while a vector only stores one object in memory

Here, the vector is seen occupying one location in memory

```
library(lobstr)
x <- letters
ref(x)
```

```
[1:0x60000174ca00] <chr>
```

While the list occupies several locations. Note also that `x` and `letters` are references to the same object in memory

```
l <- list(x, LETTERS, letters)
ref(l)
```

```
 [1:0x7fb3b89531d8] <list>
[2:0x60000174ca00] <chr>
[3:0x60000174c900] <chr>
[2:0x60000174ca00]
```

2. Why do you need to use `unlist()` to convert a list to an atomic vector? Why doesn't `as.vector()` work?

Since lists are types of vectors, the list itself will be returned when passed to `as.vector()`. This is confirmed per the documentation at `?as.vector`.

3. Compare and contrast `c()` and `unlist()` when combining a date and date-time into a single vector.

```r
d <- Sys.Date()
dt <- Sys.time()
```

- `c()` coerces into a Date or POSIXct variable, whichever is called first.

Date first:

```r
x1 <- c(d, dt)
str(x1)
```

```
 Date[1:2], format: "2022-07-09" "2022-07-09"
```

POSIXct first:

```r
x2 <- c(dt, d)
str(x2)
```

```
 POSIXct[1:2], format: "2022-07-09 10:42:55" "2022-07-08 20:00:00"
```

- `unlist()` takes a list and returns the atomic components only, so it coerces both elements into a double and returns the resulting atomic vector.

```r
y <- unlist(list(d, dt))
str(y)
```

```
 num [1:2] 1.92e+04 1.66e+09
```

2.4 3.6: Data frames and tibbles

2.4.1 Notes

Data frames and tibbles are two important S3 vectors built on lists

- Data frames are a named list of vectors with three attributes:
  - `names` for column names
  - `row.names` for row names
  - `class`, “data.frame”

- Unlike regular lists, data frames have a requirement that all vector elements have the same `NROW()`
  - Most of the time, this is equivalent to saying all columns must have the same length. However, as we'll see later, data frames can be columns, so stating the requirement in terms of `NROW()` is necessary to accommodate these cases.
  - Gives data frames the same properties as matrices
- Tibbles are a modern "equivalent" to a data.frame, provided by the `tibble` package.

```r
library(tibble)
t <- tibble()
attributes(t)
```

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

$row.names
integer(0)

$names
character(0)
```

2.4.1.1 Creating

- `data.frame()` to create data frames
- `tibble::tibble()` to create tibbles

Differences in creation between tibbles and data frames:

- Tibbles never coerce vectors
  - A common need is to suppress string to factor coercion in `data.frame()` by setting `stringsAsFactors` to `FALSE`
- Tibbles surround non-syntactic names with ` rather than transforming them
- Data frames recycle inputs that are an integer multiple of the longest vector, while tibbles only recycle vectors that are of length 1
- Tibbles allow you to refer to created variables during construction

2.4.1.2 Row names

A character vector can be supplied to label the "rows" of a data frame, in two ways:

- The `row.names` argument in `data.frame()`
- By calling `rownames()`

Tibbles do not support row names for three main reasons - Row names are stored differently from data - They only work when rows can be identified via a single string - They *must* be unique, so repetition/resampling results in new row names

Row names can be converted to a column in a tibble using `rownames_to_column()` or the `rownames` argument in `as_tibble()`

2.4.1.3 Printing

Four main differences between the `print()` output for a data frames and for tibbles:

- Tibbles show the first 10 rows and all columns that fit on screen
- Columns are labelled with its abbreviated type
- Wide columns are truncated
- Color is used when supported to (de)emphasize information

2.4.1.4 Subsetting

Discussed more in Chapter 4, you can subset data frames and tibbles like a 1-D list or a 2-D matrix.

Tibbles modify two undesirable properties of data frames:

- Data frames will return a vector for `df[, x]` if x is a length one vector, and a data frame if x is of length > 1, unless you specify `df[, x, drop = FALSE]`
  - Tibbles always return another tibble when using `[`
  - Subsetting a single column from a tibble using `[`, however, can cause an issue with some legacy code that expects an atomic vector when calling `df[, "col"]`. Use `df[["col"]]` to unambiguously return the desired column as an atomic vector whether subsetting a data frame or tibble

```r
df1 <- data.frame(xyz = "a")
df2 <- tibble(xyz = "a")
x <- "xyz"
df1[, x]
```

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

```
df1[, x, drop = FALSE]
```

```
  xyz
1   a
```

```
df2[, x]
```

```
# A tibble: 1 x 1
  xyz
  <chr>
1 a
```

- When using $ data frames will return any variable that starts with the input

```
str(df1$x)
```

```
 chr "a"
```

Tibbles only return exact matches with `$`

```
str(df2$x)
```

```
Warning: Unknown or uninitialised column: `x`.
```

```
 NULL
```

2.4.1.5 Testing and coercing

- Check with `is.data.frame()` or `is_tibble()`
- Coerce with `as.data.frame()` or `as_tibble()`

2.4.1.6 List columns

Since data frames are lists of vectors, a data frame can have a column as a list.

Adding a list column involves an extra step in data frames:

```
# Either after the data frame is created
d <- data.frame(a = 1:3)
d$b <- list(4:6)
```

Lists are fully supported in tibbles

```
d <- tibble(
  a = 1:3,
  b = list(4:6)
)
```

2.4.1.7 Matrix and data frame columns

Extending the length requirement for data frames (that all columns must be of the same length), it's actually the `NROW()` of each column that needs to match. Because of this, data frames and matrices can be included as columns in a data frame.

Just as with list columns, it must be added after creation or wrapped in `I()`. Note that wrapping it in `I()` adds a `class` "AsIs"

```
d0 <- data.frame(a = 1:3, b = 4:6)
d <- data.frame(x = -2:0, y = I(d0))
str(d)
```

```
'data.frame':	3 obs. of  2 variables:
 $ x: int  -2 -1 0
 $ y:Classes 'AsIs' and 'data.frame':	3 obs. of  2 variables:
  ..$ a: int  1 2 3
  ..$ b: int  4 5 6
```

Many functions that work with columns assume that all columns are vectors, so use with caution.

2.4.2 Exercises

1. *Can you have a data frame with zero rows? What about zero columns?*

You can create an empty data frame with zero rows and zero columns.

```
d <- data.frame()
str(d)
```

```
'data.frame':	0 obs. of  0 variables
```

You can add an empty row, but its value is inaccessible

```
d[1, ] <- 1L
d[1, ]
```

```
data frame with 0 columns and 1 row
```

Same outcome for row without a column in a tibble

```
d <- tibble()
d[1, ] <- 1L
d[1, ]
```

```
# A tibble: 1 x 0
```

You can add an empty columns during or after creation

```
d <- data.frame(x = character())
d$y <- vector("list")
str(d)
```

```
'data.frame':	0 obs. of  2 variables:
 $ x: chr 
 $ y: list()
```

2. *What happens if you attempt to set rownames that are not unique?*

R will throw an error

```
data.frame(a = 1:3, row.names = rep("x", 3))
```

```
Error in data.frame(a = 1:3, row.names = rep("x", 3)) : 
  duplicate row.names: x
NULL
```

3. *If `df` is a data frame, what can you say about `t(df)`, and `t(t(df))`? Perform some experiments, making sure to try different column types.*

If `df` can behave like a matrix, `t()` will operate as expected

```
d <- data.frame(a = 1:2, b = 3:4)
t(d)
```

```
  [,1] [,2]
a    1    2
b    3    4
```

```
t(t(d))
```

```
     a b
[1,] 1 3
[2,] 2 4
```

Prior to transposing, `t()` coerces `df` to a matrix. Non-atomic vectors are coerced by `as.vector()`. For lists, `as.vector()` returns the list, so transposition occurs at the list-element level.

Looking at the matrix object created as a first step, we see that the matrix preserved all variable types when coercing a list column into a matrix column. Remember that matrices are two dimensional vectors

```
d <- data.frame(a = 1:3, b = I(list("4", 5, 6L)))
str(as.matrix(d))
```

```
List of 6
 $ : int 1
 $ : int 2
 $ : int 3
 $ : chr "4"
 $ : num 5
 $ : int 6
 - attr(*, "dim")= int [1:2] 3 2
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:2] "a" "b"
```

So `t()` works as “expected” with list columns

```
t(d)
```

```
  [,1] [,2] [,3]
a 1    2    3
b "4"  5    6
```

```
t(t(d))
```

```
     a b
[1,] 1 "4"
[2,] 2 5
[3,] 3 6
```

For data frame columns, `as.matrix()` will combine the data frame columns (and any nested data frame columns) with the containing data frame.

```
d <- data.frame(a = 1:3, b = 4:6)
d$c <- data.frame(x = 101:103, y = 104:106)
d0 <- data.frame(z = 1001:1003)
d0$zz <- data.frame(z0 = 1:3, z1 = 4:6)
d$d <- d0
as.matrix(d)
```

```
     a b c.x c.y  d.z d.zz.z0 d.zz.z1
[1,] 1 4 101 104 1001       1       4
[2,] 2 5 102 105 1002       2       5
[3,] 3 6 103 106 1003       3       6
```

`t()` works with data frame columns normally after combining

```
t(d)
```

```
        [,1] [,2] [,3]
a          1    2    3
b          4    5    6
c.x      101  102  103
c.y      104  105  106
d.z     1001 1002 1003
d.zz.z0    1    2    3
d.zz.z1    4    5    6
```

```
t(t(d))
```

```
     a b c.x c.y  d.z d.zz.z0 d.zz.z1
[1,] 1 4 101 104 1001       1       4
[2,] 2 5 102 105 1002       2       5
[3,] 3 6 103 106 1003       3       6
```

4. What does `as.matrix()` do when applied to a data frame with columns of different types? How does it differ from `data.matrix()`?

The behavior for `as.matrix()` is described in answer 3.

`data.matrix()` converts all variables in a data frame to numeric via `as.numeric()` prior to combining them, so I would expect odd behavior when converting non numeric variable types

Logical and factor columns are coerced to integers. Character columns are converted first to factors, then to integers

```
a <- letters[1:3]
as.integer(as.factor(a))
```

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

So `data.matrix()` will use the resulting integer column for the resulting matrix

```
b <- c(TRUE, TRUE, FALSE)
c <- factor(LETTERS[1:3])
d <- data.frame(a, b, c)
data.matrix(d)
```

```
     a b c
[1,] 1 1 1
[2,] 2 1 2
[3,] 3 0 3
```

Any non numeric column that is *not* a logical, factor, or character column is converted to a numeric column via `as.numeric()`. `as.numeric()` only works on atomic vectors, so it will convert each list element at the atomic level

```r
b <- list(a = 1, b = "2", c = 3L)
d <- data.frame(b = b)
data.matrix(d)
```

```
     b.a b.b b.c
[1,]   1   1   3
```

Similar application for data frames, being lists themselves

```r
c <- data.frame(c1 = 4:6, c2 = 7:9)
d <- data.frame(a = 1:3, c = c)
data.matrix(d)
```

```
     a c.c1 c.c2
[1,] 1    4    7
[2,] 2    5    8
[3,] 3    6    9
```

2.5 3.7: NULL

2.5.1 Notes

- `NULL` is a data structure with a unique type, "NULL"

```r
typeof(NULL)
```

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

- `NULL` is always length zero

```r
length(NULL)
```

```
[1] 0
```

- `NULL` cannot have any attributes

```r
x <- NULL
attr(x, a) <- 1L
```

```
Error in attr(x, a) <- 1L : attempt to set an attribute on NULL
NULL
```

Two common uses of `NULL`:

- To represent an empty vector
- To represent an absent vector
    - `NULL` is often used as a default function value, to signify that that value is not needed in the function.
    - `NA`, in contrast, signifies that the element of a vector is absent, not the vector itself