2

Data structures

This chapter summarises the most important data structures in base R. You've probably used many (if not all) of them before, but you may not have thought deeply about how they are interrelated. In this brief overview, I won't discuss individual types in depth. Instead, I'll show you how they fit together as a whole. If you need more details, you can find them in R's documentation.

R's base data structures can be organised by their dimensionality (1d, 2d, or nd) and whether they're homogeneous (all contents must be of the same type) or heterogeneous (the contents can be of different types). This gives rise to the five data types most often used in data analysis:

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Almost all other objects are built upon these foundations. In Chapter 7 you'll see how more complicated objects are built of these simple pieces. Note that R has no 0-dimensional, or scalar types. Individual numbers or strings, which you might think would be scalars, are actually vectors of length one.

Given an object, the best way to understand what data structures it's composed of is to use `str()`. `str()` is short for structure and it gives a compact, human readable description of any R data structure.

Quiz

Take this short quiz to determine if you need to read this chapter. If the answers quickly come to mind, you can comfortably skip this chapter. You can check your answers in Section 2.5.

1. What are the three properties of a vector, other than its contents?
2. What are the four common types of atomic vectors? What are the two rare types?
3. What are attributes? How do you get them and set them?
4. How is a list different from an atomic vector? How is a matrix different from a data frame?
5. Can you have a list that is a matrix? Can a data frame have a column that is a matrix?

Outline

- Section 2.1 introduces you to atomic vectors and lists, R's 1d data structures.
- Section 2.2 takes a small detour to discuss attributes, R's flexible metadata specification. Here you'll learn about factors, an important data structure created by setting attributes of an atomic vector.
- Section 2.3 introduces matrices and arrays, data structures for storing 2d and higher dimensional data.
- Section 2.4 teaches you about the data frame, the most important data structure for storing data in R. Data frames combine the behaviour of lists and matrices to make a structure ideally suited for the needs of statistical data.

2.1 Vectors

The basic data structure in R is the vector. Vectors come in two flavours: atomic vectors and lists. They have three common properties:

- Type, `typeof()`, what it is.

- Length, `length()`, how many elements it contains.
- Attributes, `attributes()`, additional arbitrary metadata.

They differ in the types of their elements: all elements of an atomic vector must be the same type, whereas the elements of a list can have different types.

NB: `is.vector()` does not test if an object is a vector. Instead it returns `TRUE` only if the object is a vector with no attributes apart from names. Use `is.atomic(x) || is.list(x)` to test if an object is actually a vector.

2.1.1 Atomic vectors

There are four common types of atomic vectors that I'll discuss in detail: logical, integer, double (often called numeric), and character. There are two rare types that I will not discuss further: complex and raw.

Atomic vectors are usually created with `c()`, short for combine:

```
dbl_var <- c(1, 2.5, 4.5)
# With the L suffix, you get an integer rather than a double
int_var <- c(1L, 6L, 10L)
# Use TRUE and FALSE (or T and F) to create logical vectors
log_var <- c(TRUE, FALSE, T, F)
chr_var <- c("these are", "some strings")
```

Atomic vectors are always flat, even if you nest `c()`'s:

```
c(1, c(2, c(3, 4)))
#> [1] 1 2 3 4
# the same as
c(1, 2, 3, 4)
#> [1] 1 2 3 4
```

Missing values are specified with `NA`, which is a logical vector of length 1. `NA` will always be coerced to the correct type if used inside `c()`, or you can create `NA`s of a specific type with `NA_real_` (a double vector), `NA_integer_` and `NA_character_`.

2.1.1.1 Types and tests

Given a vector, you can determine its type with `typeof()`, or check if it's a specific type with an "is" function: `is.character()`, `is.double()`, `is.integer()`, `is.logical()`, or, more generally, `is.atomic()`.

```
int_var <- c(1L, 6L, 10L)
typeof(int_var)
#> [1] "integer"
is.integer(int_var)
#> [1] TRUE
is.atomic(int_var)
#> [1] TRUE

dbl_var <- c(1, 2.5, 4.5)
typeof(dbl_var)
#> [1] "double"
is.double(dbl_var)
#> [1] TRUE
is.atomic(dbl_var)
#> [1] TRUE
```

NB: `is.numeric()` is a general test for the "numberliness" of a vector and returns `TRUE` for both integer and double vectors. It is not a specific test for double vectors, which are often called numeric.

```
is.numeric(int_var)
#> [1] TRUE
is.numeric(dbl_var)
#> [1] TRUE
```

2.1.1.2 Coercion

All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be **coerced** to the most flexible type. Types from least to most flexible are: logical, integer, double, and character.

For example, combining a character and an integer yields a character:

```
str(c("a", 1))
#>  chr [1:2] "a" "1"
```

When a logical vector is coerced to an integer or double, `TRUE` becomes 1 and `FALSE` becomes 0. This is very useful in conjunction with `sum()` and `mean()`

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
#> [1] 0 0 1

# Total number of TRUEs
sum(x)
#> [1] 1

# Proportion that are TRUE
mean(x)
#> [1] 0.3333
```

Coercion often happens automatically. Most mathematical functions (`+`, `log`, `abs`, etc.) will coerce to a double or integer, and most logical operations (`&`, `|`, `any`, etc) will coerce to a logical. You will usually get a warning message if the coercion might lose information. If confusion is likely, explicitly coerce with `as.character()`, `as.double()`, `as.integer()`, or `as.logical()`.

2.1.2 Lists

Lists are different from atomic vectors because their elements can be of any type, including lists. You construct lists by using `list()` instead of `c()`:

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
#> List of 4
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

Lists are sometimes called **recursive** vectors, because a list can contain other lists. This makes them fundamentally different from atomic vectors.

```
x <- list(list(list(list())))
str(x)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : list()
is.recursive(x)
#> [1] TRUE
```

`c()` will combine several lists into one. If given a combination of atomic vectors and lists, `c()` will coerce the vectors to list before combining them. Compare the results of `list()` and `c()`:

```
x <- list(list(1, 2), c(3, 4))
y <- c(list(1, 2), c(3, 4))
str(x)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
str(y)
#> List of 4
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```

The `typeof()` a list is `list`. You can test for a list with `is.list()` and coerce to a list with `as.list()`. You can turn a list into an atomic vector with `unlist()`. If the elements of a list have different types, `unlist()` uses the same coercion rules as `c()`.

Lists are used to build up many of the more complicated data structures in R. For example, both data frames (described in Section 2.4) and linear models objects (as produced by `lm()`) are lists:

```
is.list(mtcars)
#> [1] TRUE

mod <- lm(mpg ~ wt, data = mtcars)
is.list(mod)
#> [1] TRUE
```

2.1.3 Exercises

1. What are the six types of atomic vector? How does a list differ from an atomic vector?
2. What makes `is.vector()` and `is.numeric()` fundamentally different to `is.list()` and `is.character()`?
3. Test your knowledge of vector coercion rules by predicting the output of the following uses of `c()`:

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

4. Why do you need to use `unlist()` to convert a list to an atomic vector? Why doesn't `as.vector()` work?
5. Why is `1 == "1"` true? Why is `-1 < FALSE` true? Why is `"one" < 2` false?
6. Why is the default missing value, `NA`, a logical vector? What's special about logical vectors? (Hint: think about `c(FALSE, NA_character_)`.)

2.2 Attributes

All objects can have arbitrary additional attributes, used to store metadata about the object. Attributes can be thought of as a named list (with unique names). Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`.

```
y <- 1:10
attr(y, "my_attribute") <- "This is a vector"
attr(y, "my_attribute")
#> [1] "This is a vector"
str(attributes(y))
#> List of 1
#>  $ my_attribute: chr "This is a vector"
```

The `structure()` function returns a new object with modified attributes:

```
structure(1:10, my_attribute = "This is a vector")
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> attr(,"my_attribute")
#> [1] "This is a vector"
```

By default, most attributes are lost when modifying a vector:

```
attributes(y[1])
#> NULL
attributes(sum(y))
#> NULL
```

The only attributes not lost are the three most important:

- Names, a character vector giving each element a name, described in Section 2.2.0.1.
- Dimensions, used to turn vectors into matrices and arrays, described in Section 2.3.
- Class, used to implement the S3 object system, described in Section 7.2.

Each of these attributes has a specific accessor function to get and set values. When working with these attributes, use `names(x)`, `class(x)`, and `dim(x)`, not `attr(x, "names")`, `attr(x, "class")`, and `attr(x, "dim")`.

2.2.0.1 Names

You can name a vector in three ways:

- When creating it: `x <- c(a = 1, b = 2, c = 3)`.
- By modifying an existing vector in place: `x <- 1:3; names(x) <- c("a", "b", "c")`.
- By creating a modified copy of a vector: `x <- setNames(1:3, c("a", "b", "c"))`.

Names don't have to be unique. However, character subsetting, described in Section 3.4.1, is the most important reason to use names and it is most useful when the names are unique.

Not all elements of a vector need to have a name. If some names are missing, `names()` will return an empty string for those elements. If all names are missing, `names()` will return `NULL`.

```
y <- c(a = 1, 2, 3)
names(y)
#> [1] "a" ""  ""

z <- c(1, 2, 3)
names(z)
#> NULL
```

You can create a new vector without names using `unname(x)`, or remove names in place with `names(x) <- NULL`.

2.2.1 Factors

One important use of attributes is to define factors. A factor is a vector that can contain only predefined values, and is used to store categorical data. Factors are built on top of integer vectors using two attributes: the `class()`, "factor", which makes them behave differently from regular integer vectors, and the `levels()`, which defines the set of allowed values.

```
x <- factor(c("a", "b", "b", "a"))
x
#> [1] a b b a
#> Levels: a b
class(x)
#> [1] "factor"
levels(x)
#> [1] "a" "b"

# You can't use values that are not in the levels
x[2] <- "c"
#> Warning: invalid factor level, NA generated
x
#> [1] a    <NA> b    a
#> Levels: a b

# NB: you can't combine factors
c(factor("a"), factor("b"))
#> [1] 1 1
```

Factors are useful when you know the possible values a variable may take, even if you don't see all values in a given dataset. Using a factor

instead of a character vector makes it obvious when some groups contain no observations:

```
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
#> sex_char
#> m
#> 3
table(sex_factor)
#> sex_factor
#> m f
#> 3 0
```

Sometimes when a data frame is read directly from a file, a column you'd thought would produce a numeric vector instead produces a factor. This is caused by a non-numeric value in the column, often a missing value encoded in a special way like `.` or `-`. To remedy the situation, coerce the vector from a factor to a character vector, and then from a character to a double vector. (Be sure to check for missing values after this process.) Of course, a much better plan is to discover what caused the problem in the first place and fix that; using the `na.strings` argument to `read.csv()` is often a good place to start.

```
# Reading in "text" instead of from a file here:
z <- read.csv(text = "value\n12\n1\n.\n9")
typeof(z$value)
#> [1] "integer"
as.double(z$value)
#> [1] 3 2 1 4
# Oops, that's not right: 3 2 1 4 are the levels of a factor,
# not the values we read in!
class(z$value)
#> [1] "factor"
# We can fix it now:
as.double(as.character(z$value))
#> Warning: NAs introduced by coercion
#> [1] 12  1 NA  9
# Or change how we read it in:
z <- read.csv(text = "value\n12\n1\n.\n9", na.strings=".")
typeof(z$value)
```

```
#> [1] "integer"
class(z$value)
#> [1] "integer"
z$value
#> [1] 12  1 NA  9
# Perfect! :)
```

Unfortunately, most data loading functions in R automatically convert character vectors to factors. This is suboptimal, because there's no way for those functions to know the set of all possible levels or their optimal order. Instead, use the argument `stringsAsFactors = FALSE` to suppress this behaviour, and then manually convert character vectors to factors using your knowledge of the data. A global option, `options(stringsAsFactors = FALSE)`, is available to control this behaviour, but I don't recommend using it. Changing a global option may have unexpected consequences when combined with other code (either from packages, or code that you're `source()`ing), and global options make code harder to understand because they increase the number of lines you need to read to understand how a single line of code will behave.

While factors look (and often behave) like character vectors, they are actually integers. Be careful when treating them like strings. Some string methods (like `gsub()` and `grepl()`) will coerce factors to strings, while others (like `nchar()`) will throw an error, and still others (like `c()`) will use the underlying integer values. For this reason, it's usually best to explicitly convert factors to character vectors if you need string-like behaviour. In early versions of R, there was a memory advantage to using factors instead of character vectors, but this is no longer the case.

2.2.2 Exercises

1. 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.)

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

   ```
   f1 <- factor(letters)
   levels(f1) <- rev(levels(f1))
   ```

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

```
f2 <- rev(factor(letters))

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

2.3 Matrices and arrays

Adding a `dim()` attribute to an atomic vector allows it to behave like a multi-dimensional **array**. A special case of the array is the **matrix**, which has two dimensions. Matrices are used commonly as part of the mathematical machinery of statistics. Arrays are much rarer, but worth being aware of.

Matrices and arrays are created with `matrix()` and `array()`, or by using the assignment form of `dim()`:

```
# Two scalar arguments to specify rows and columns
a <- matrix(1:6, ncol = 3, nrow = 2)
# One vector argument to describe all dimensions
b <- array(1:12, c(2, 3, 2))

# You can also modify an object in place by setting dim()
c <- 1:6
dim(c) <- c(3, 2)
c
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
dim(c) <- c(2, 3)
c
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

`length()` and `names()` have high-dimensional generalisations:

- `length()` generalises to `nrow()` and `ncol()` for matrices, and `dim()` for arrays.

- `names()` generalises to `rownames()` and `colnames()` for matrices, and `dimnames()`, a list of character vectors, for arrays.

```
length(a)
#> [1] 6
nrow(a)
#> [1] 2
ncol(a)
#> [1] 3
rownames(a) <- c("A", "B")
colnames(a) <- c("a", "b", "c")
a
#>   a b c
#> A 1 3 5
#> B 2 4 6

length(b)
#> [1] 12
dim(b)
#> [1] 2 3 2
dimnames(b) <- list(c("one", "two"), c("a", "b", "c"), c("A", "B"))
b
#> , , A
#>
#>     a b c
#> one 1 3 5
#> two 2 4 6
#>
#> , , B
#>
#>     a  b  c
#> one 7  9 11
#> two 8 10 12
```

`c()` generalises to `cbind()` and `rbind()` for matrices, and to `abind()` (provided by the `abind` package) for arrays. You can transpose a matrix with `t()`; the generalised equivalent for arrays is `aperm()`.

You can test if an object is a matrix or array using `is.matrix()` and `is.array()`, or by looking at the length of the `dim()`. `as.matrix()` and `as.array()` make it easy to turn an existing vector into a matrix or array.

Vectors are not the only 1-dimensional data structure. You can have matrices with a single row or single column, or arrays with a single

dimension. They may print similarly, but will behave differently. The differences aren't too important, but it's useful to know they exist in case you get strange output from a function (`tapply()` is a frequent offender). As always, use `str()` to reveal the differences.

```
str(1:3)                   # 1d vector
#>  int [1:3] 1 2 3
str(matrix(1:3, ncol = 1)) # column vector
#>  int [1:3, 1] 1 2 3
str(matrix(1:3, nrow = 1)) # row vector
#>  int [1, 1:3] 1 2 3
str(array(1:3, 3))         # "array" vector
#>  int [1:3(1d)] 1 2 3
```

While atomic vectors are most commonly turned into matrices, the dimension attribute can also be set on lists to make list-matrices or list-arrays:

```
l <- list(1:3, "a", TRUE, 1.0)
dim(l) <- c(2, 2)
l
#>      [,1]      [,2]
#> [1,] Integer,3 TRUE
#> [2,] "a"       1
```

These are relatively esoteric data structures, but can be useful if you want to arrange objects into a grid-like structure. For example, if you're running models on a spatio-temporal grid, it might be natural to preserve the grid structure by storing the models in a 3d array.

2.3.1 Exercises

1. What does `dim()` return when applied to a vector?
2. If `is.matrix(x)` is `TRUE`, what will `is.array(x)` return?
3. How would you describe the following three objects? What makes them different to `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))
   ```

2.4 Data frames

A data frame is the most common way of storing data in R, and if used systematically (http://vita.had.co.nz/papers/tidy-data.pdf) makes data analysis easier. Under the hood, a data frame is a list of equal-length vectors. This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list. This means that a data frame has `names()`, `colnames()`, and `rownames()`, although `names()` and `colnames()` are the same thing. The `length()` of a data frame is the length of the underlying list and so is the same as `ncol()`; `nrow()` gives the number of rows.

As described in Chapter 3, you can subset a data frame like a 1d structure (where it behaves like a list), or a 2d structure (where it behaves like a matrix).

2.4.1 Creation

You create a data frame using `data.frame()`, which takes named vectors as input:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Beware `data.frame()`'s default behaviour which turns strings into factors. Use `stringAsFactors = FALSE` to suppress this behaviour:

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE)
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

2.4.2 Testing and coercion

Because a `data.frame` is an S3 class, its type reflects the underlying vector used to build it: the list. To check if an object is a data frame, use `class()` or test explicitly with `is.data.frame()`:

```
typeof(df)
#> [1] "list"
class(df)
#> [1] "data.frame"
is.data.frame(df)
#> [1] TRUE
```

You can coerce an object to a data frame with `as.data.frame()`:

- A vector will create a one-column data frame.
- A list will create one column for each element; it's an error if they're not all the same length.
- A matrix will create a data frame with the same number of columns and rows.

2.4.3 Combining data frames

You can combine data frames using `cbind()` and `rbind()`:

```
cbind(df, data.frame(z = 3:1))
#>   x y z
#> 1 1 a 3
#> 2 2 b 2
#> 3 3 c 1
rbind(df, data.frame(x = 10, y = "z"))
#>    x y
#> 1  1 a
#> 2  2 b
#> 3  3 c
#> 4 10 z
```

When combining column-wise, the number of rows must match, but row names are ignored. When combining row-wise, both the number

and names of columns must match. Use `plyr::rbind.fill()` to combine data frames that don't have the same columns.

It's a common mistake to try and create a data frame by `cbind()`ing vectors together. This doesn't work because `cbind()` will create a matrix unless one of the arguments is already a data frame. Instead use `data.frame()` directly:

```
bad <- data.frame(cbind(a = 1:2, b = c("a", "b")))
str(bad)
#> 'data.frame':    2 obs. of  2 variables:
#>  $ a: Factor w/ 2 levels "1","2": 1 2
#>  $ b: Factor w/ 2 levels "a","b": 1 2
good <- data.frame(a = 1:2, b = c("a", "b"),
  stringsAsFactors = FALSE)
str(good)
#> 'data.frame':    2 obs. of  2 variables:
#>  $ a: int  1 2
#>  $ b: chr  "a" "b"
```

The conversion rules for `cbind()` are complicated and best avoided by ensuring all inputs are of the same type.

2.4.4 Special columns

Since a data frame is a list of vectors, it is possible for a data frame to have a column that is a list:

```
df <- data.frame(x = 1:3)
df$y <- list(1:2, 1:3, 1:4)
df
#>   x          y
#> 1 1       1, 2
#> 2 2    1, 2, 3
#> 3 3 1, 2, 3, 4
```

However, when a list is given to `data.frame()`, it tries to put each item of the list into its own column, so this fails:

```
data.frame(x = 1:3, y = list(1:2, 1:3, 1:4))
#> Error: arguments imply differing number of rows: 2, 3, 4
```

A workaround is to use `I()`, which causes `data.frame()` to treat the list as one unit:

```
dfl <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))
str(dfl)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y:List of 3
#>   ..$ : int  1 2
#>   ..$ : int  1 2 3
#>   ..$ : int  1 2 3 4
#>   ..- attr(*, "class")= chr "AsIs"
dfl[2, "y"]
#> [[1]]
#> [1] 1 2 3
```

`I()` adds the `AsIs` class to its input, but this can usually be safely ignored.

Similarly, it's also possible to have a column of a data frame that's a matrix or array, as long as the number of rows matches the data frame:

```
dfm <- data.frame(x = 1:3, y = I(matrix(1:9, nrow = 3)))
str(dfm)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: 'AsIs' int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
dfm[2, "y"]
#>      [,1] [,2] [,3]
#> [1,]    2    5    8
```

Use list and array columns with caution: many functions that work with data frames assume that all columns are atomic vectors.

2.4.5 Exercises

1. What attributes does a data frame possess?
2. What does `as.matrix()` do when applied to a data frame with columns of different types?
3. Can you have a data frame with 0 rows? What about 0 columns?

2.5 Answers

1. The three properties of a vector are type, length, and attributes.
2. The four common types of atomic vector are logical, integer, double (sometimes called numeric), and character. The two rarer types are complex and raw.
3. Attributes allow you to associate arbitrary additional metadata to any object. You can get and set individual attributes with `attr(x, "y")` and `attr(x, "y") <- value`; or get and set all attributes at once with `attributes()`.
4. The elements of a list can be any type (even a list); the elements of an atomic vector are all of the same type. Similarly, every element of a matrix must be the same type; in a data frame, the different columns can have different types.
5. You can make "list-array" by assuming dimensions to a list. You can make a matrix a column of a data frame with `df$x <- matrix()`, or using `I()` when creating a new data frame `data.frame(x = I(matrix()))`.