One-Dimensional Data in R

PS239T

03 February, 2019

Intro to Data Structures

To make the best use of the R language, you'll need a strong understanding of the basic data types and data structures and how to operate on those. R is an **object-oriented** language, so the importance of this cannot be understated.

It is **critical** to understand because these are the objects you will manipulate on a day-to-day basis in R, and they are not always as easy to work with as they sound at the outset. Dealing with object conversions is one of the most common sources of frustration for beginners.

> To understand computations in R, two slogans are helpful: - Everything that exists is an object. - Everything that happens is a function call.
>
> ___John Chambers___the creator of S (the mother of R)

Outline

In this markdown file, we cover the types of one-dimensional objects (vectors) that exist in R and their attributes.

1. Main Classes introduces you to R's one-dimensional or atomic classes and data structures. R has five basic atomic classes: logical, integer, numeric, complex, character. Social scientists don't use complex class. (Also, remember that we rarely use trigonometry.)
2. Attributes 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. R has many data structures: vector, list, matrix, data frame, factors, tables.

1. Main Classes of Vectors

1a. Atomic Classes

`R`'s main atomic classes are:

- character (or a "string" in Python and Stata)
- numeric (integer or float)
- integer (just integer)
- logical (booleans)

Example	Type
"a", "swc"	character
2, 15.5	numeric
2 (Must add a `L` at end to denote integer)	integer
`TRUE`, `FALSE`	logical

Like Python, R is dynamically typed. There are a few differences in terminology, however, that are pertinent.

- First, "types" in Python are referred to as "classes" in R.
- Second, R has some different names for the types string, integer, and float — specifically **character**, **integer** (not different), and **numeric**. Because there is no "float" class in R, users tend to default to the "numeric" class when they want to work with numerical data.

The function for recovering object classes is `class()`. L suffix to qualify any number with the intent of making it an explicit integer. See more from the R language definition.

```
class(3)
```

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

```
class(3L)
```

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

```
class("Three")
```

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

```
class(T)
```

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

1b. Data Structures

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	

Each data structure has its own specifications and behavior. For our purposes, an important thing to remember is that R is always **faster** (more efficient) working with homogenous (**vectorized**) data.

1c. Vector properties

Vectors have three common properties:

- Class, `class()`, or what type of object it is (same as `type()` in Python).
- Length, `length()`, how many elements it contains (same as `len()` in Python).
- 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.

1d. Creating different types of atomic vectors

Remember, there are four common types of vectors: * `logical` * `integer` * `numeric` (same as `double`) * `character`.

You can create an empty vector with `vector()` (By default the mode is `logical`. You can be more explicit as shown in the examples below.) It is more common to use direct constructors such as `character()`, `numeric()`, etc.

```
x <- vector()
# with a length and type
vector("character", length = 10)
```

```
##  [1] "" "" "" "" "" "" "" "" "" ""
```

```
character(5) ## character vector of length 5
```

```
## [1] "" "" "" "" ""
```

```
numeric(5)
```

```
## [1] 0 0 0 0 0
```

```
logical(5)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE
```

Atomic vectors are usually created with `c()`, which is short for concatenate:

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

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

```
length(x)
```

```
## [1] 3
```

`x` is a numeric vector. These are the most common kind. You can also have logical vectors.

```
y <- c(TRUE, TRUE, FALSE, FALSE)
```

Finally you can have character vectors:

```
kim_family <- c("Jae", "Sun", "Jane")

is.integer(kim_family) # integer?
```

```
## [1] FALSE
```

```
is.character(kim_family) # character?
```

```
## [1] TRUE
```

```
is.atomic(kim_family) # atomic?
```

```
## [1] TRUE
```

```
typeof(kim_family) # what's the type?
```

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

Short exercise: Create and examine your vector

Create a character vector called `fruit` that contain 4 of your favorite fruits. Then evaluate its structure using the commands below.

```
# First create your fruit vector
# YOUR CODE HERE
```

```
# Examine your vector
length(fruit)
class(fruit)
str(fruit)
```

Add elements

You can add elements to the end of a vector by passing the original vector into the `c` function, like so:

```
z <- c("Beyonce", "Kelly", "Michelle", "LeToya")
z <- c(z, "Farrah")
z
```

```
## [1] "Beyonce"  "Kelly"    "Michelle" "LeToya"   "Farrah"
```

More examples of vectors

```
x <- c(0.5, 0.7)
x <- c(TRUE, FALSE)
x <- c("a", "b", "c", "d", "e")
x <- 9:100
```

You can also create vectors as a sequence of numbers

```
series <- 1:10
seq(10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1, 10, by = 0.1)
```

```
##  [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3
## [15]  2.4  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7
## [29]  3.8  3.9  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1
## [43]  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5
## [57]  6.6  6.7  6.8  6.9  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9
## [71]  8.0  8.1  8.2  8.3  8.4  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3
## [85]  9.4  9.5  9.6  9.7  9.8  9.9 10.0
```

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
```

Types and Tests

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

```
char_var <- c("harry", "sally")
class(char_var)
```

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

```
is.character(char_var)
```

```
## [1] TRUE
```

```
is.atomic(char_var)
```

```
## [1] TRUE
```

```
num_var <- c(1, 2.5, 4.5)
class(num_var)
```

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

```
is.numeric(num_var)
```

```
## [1] TRUE
```

```
is.atomic(num_var)
```

```
## [1] TRUE
```

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.

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"
```

Guess what the following do without running them first

```
c(1.7, "a")
c(TRUE, 2)
c("a", TRUE)
```

Notice that 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] 0.3333333
```

Coercion often happens automatically. This is called implicit coercion. Most mathematical functions (`+`, `log`, `abs`, etc.) will coerce to a numeric 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.

```
1 < "2"
```

```
## [1] TRUE
```

```
"1" > 2
```

```
## [1] FALSE
```

You can also coerce vectors explicitly coerce with `as.character()`, `as.numeric()`, `as.integer()`, or `as.logical()`. Example:

```
x <- 0:6
as.numeric(x)
```

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

```
as.logical(x)
```

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

```
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

Sometimes coercions, especially nonsensical ones, won't work.

```
x <- c("a", "b", "c")
as.numeric(x)
```

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

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

```
as.logical(x)
```

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

```
# both don't work
```

Short Exercise

```
# 1. Create a vector of a sequence of numbers between 1 to 10.

# 2. Coerce that vector into a character vector

# 3. Add the element "11" to the end of the vector

# 4. Coerce it back to a numeric vector.
```

1e. Lists

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

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

```
x <- list(1, "a", TRUE, c(4, 5, 6))
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
```

```
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 4 5 6
```

You can coerce other objects using `as.list()`. You can test for a list with `is.list()`

```
x <- 1:10
x <- as.list(x)
is.list(x)
```

```
## [1] TRUE
```

```
length(x)
```

```
## [1] 10
```

`c()` will combine several lists into one. If given a combination of atomic vectors and lists, `c()` (con**c**atenate) will coerce the vectors to lists 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
```

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()`.

```
x <- list(list(1, 2), c(3, 4))
x
```

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

```
unlist(x)
```

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

Lists are used to build up many of the more complicated data structures in R. For example, both data frames 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
```

For this reason, lists are extremely useful inside functions. You can "staple" together lots of different kinds of results into a single object that a function can return.

A list does not print to the console like a vector. Instead, each element of the list starts on a new line.

```
x.vec <- c(1,2,3)
x.list <- list(1,2,3)
x.vec
```

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

```
x.list
```

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

For lists, elements are **indexed by double brackets**. Single brackets will still return a(nother) list. (We'll talk more about subsetting and indexing in the fourth lesson.)

Exercises

1. What are the four basic types of atomic vector? How does a list differ from an atomic vector?
2. Why is `1 == "1"` true? Why is `-1 < FALSE` true? Why is `"one" < 2` false?
3. Create three vectors and then combine them into a list.
4. If `x` is a list, what is the class of `x[1]`? How about `x[[1]]`?

2. Attributes

Attributes provide additional information about the data to you, the user, and to R. We've already seen the following three attributes in action:

- Names, a character vector giving each element a name.
- Dimensions, used to turn vectors into matrices.
- Class, used to implement the S3 object system.

Each of these attributes has a specific accessor function to get and set values. When working with these attributes, use `names(x)`, `dim(x)`, and `class(x)`

2a. Names

You can name a vector when you create it:

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

You can also modifying an existing vector:

```
x <- 1:3
names(x)
```

```
## NULL
```

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

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

Names don't have to be unique. However, character subsetting, described in the next lesson, is the most important reason to use names and it is most useful when the names are unique. (For Python users: when names are unique, a vector behaves kind of like a Python dictionary key.)

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`.

2b. Factors

Factors are special vectors that represent categorical data. Factors can be ordered or unordered and are important for modelling functions such as `lm()` and `glm()` and also in plot methods.

Factors can only contain pre-defined values. Set allowed values using the `levels()` attribute.

```
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 in `[<-.factor`(`*tmp*`, 2, value = "c"): invalid factor level, NA
## generated
```

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

```
## [1] 1 1
```

Factors are pretty much integers that have labels on them. Underneath, it's really numbers (1, 2, 3...).

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

```
##  Factor w/ 2 levels "a","b": 1 2 2 1
```

They are better than using simple integer labels because factors are what are called self describing. For example, `democrat` and `republican` is more descriptive than `1`s and `2`s.

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:

```
party_char <- c("democrat", "democrat", "democrat")
party_char
```

```
## [1] "democrat" "democrat" "democrat"
```

```
party_factor <- factor(party_char, levels = c("democrat", "republican"))
party_factor
```

```
## [1] democrat democrat democrat
## Levels: democrat republican
```

```
table(party_char)
```

```
## party_char
## democrat
##        3
```

```
table(party_factor)
```

```
## party_factor
##   democrat republican
##          3          0
```

Sometimes factors can be left unordered. Example: `democrat`, `republican`.

Other times you might want factors to be ordered (or ranked). Example: `low`, `medium`, `high`.

```
x <- factor(c("low","medium","high"))
str(x)
```

```
##  Factor w/ 3 levels "high","low","medium": 2 3 1
```

```
is.ordered(x)
```

```
## [1] FALSE
```

```
x <- ordered(c("low","medium","high"), levels = c("high","medium","low"))
is.ordered(x)
```

```
## [1] TRUE
```

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.

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

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

```
is.factor(x)
```

```
## [1] FALSE
```

```
x <- as.factor(x)
x
```

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

```
c(x, "c")
```

```
## [1] "1" "2" "2" "1" "c"
```

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.

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. If this becomes a problem, use the argument `stringsAsFactors = FALSE` to suppress this behaviour, and then manually convert character vectors to factors using your knowledge of the data.

More attributes

All R 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)) # str returns a new object with modified information
```

```
## List of 1
##  $ my_attribute: chr "This is a vector"
```

Exercises

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

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

```
##  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
## Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a
```

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

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