

The `data.frame` class

Data Carpentry contributors

Learning Objectives

- understand the concept of a `data.frame`
 - use sequences
 - know how to access any element of a `data.frame`
-

What are data frames?

Data frames are the *de facto* data structure for most tabular data, and what we use for statistics and plotting.

A data frame is a collection of vectors of identical lengths. Each vector represents a column, and each vector can be of a different data type (e.g., characters, integers, factors). The `str()` function is useful to inspect the data types of the columns.

A data frame can be created by hand, but most commonly they are generated by the functions `read.csv()` or `read.table()`; in other words, when importing spreadsheets from your hard drive (or the web).

By default, when building or importing a data frame, the columns that contain characters (i.e., text) are coerced (=converted) into the `factor` data type. Depending on what you want to do with the data, you may want to keep these columns as `character`. To do so, `read.csv()` and `read.table()` have an argument called `stringsAsFactors` which can be set to `FALSE`:

```
some_data <- read.csv("data/some_file.csv", stringsAsFactors=FALSE)
```

You can also create a data frame manually with the function `data.frame()`. This function can also take the argument `stringsAsFactors`. Compare the output of these examples, and compare the difference between when the data are being read as `character`, and when they are being read as `factor`.

```
## Compare the output of these examples, and compare the difference between when
## the data are being read as `character`, and when they are being read as
## `factor`.
example_data <- data.frame(animal=c("dog", "cat", "sea cucumber", "sea urchin"),
                           feel=c("furry", "furry", "squishy", "spiny"),
                           weight=c(45, 8, 1.1, 0.8))
str(example_data)
```

```
#> 'data.frame':  4 obs. of  3 variables:
#> $ animal: Factor w/ 4 levels "cat","dog","sea cucumber",...: 2 1 3 4
#> $ feel  : Factor w/ 3 levels "furry","spiny",...: 1 1 3 2
#> $ weight: num  45 8 1.1 0.8
```

```
example_data <- data.frame(animal=c("dog", "cat", "sea cucumber", "sea urchin"),
                           feel=c("furry", "furry", "squishy", "spiny"),
                           weight=c(45, 8, 1.1, 0.8), stringsAsFactors=FALSE)
str(example_data)
```

```
#> 'data.frame': 4 obs. of 3 variables:
#> $ animal: chr "dog" "cat" "sea cucumber" "sea urchin"
#> $ feel : chr "furry" "furry" "squishy" "spiny"
#> $ weight: num 45 8 1.1 0.8
```

Challenge

1. There are a few mistakes in this hand crafted `data.frame`, can you spot and fix them? Don't hesitate to experiment!

```
author_book <- data.frame(author_first=c("Charles", "Ernst", "Theodosius"),
                           author_last=c(Darwin, Mayr, Dobzhansky),
                           year=c(1942, 1970))
```

2. Can you predict the class for each of the columns in the following example? Check your guesses using `str(country_climate)`:

- Are they what you expected? Why? Why not?
- What would have been different if we had added `stringsAsFactors = FALSE` to this call?
- What would you need to change to ensure that each column had the accurate data type?

```
country_climate <- data.frame(country=c("Canada", "Panama", "South Africa", "Australia"),
                               climate=c("cold", "hot", "temperate", "hot/temperate"),
                               temperature=c(10, 30, 18, "15"),
                               northern_hemisphere=c(TRUE, TRUE, FALSE, "FALSE"),
                               has_kangaroo=c(FALSE, FALSE, FALSE, 1))
```

```
r ## Answers ## * missing quotations around the first names of the authors ## * the year
column is missing one value, 1859 (the year of publication of ## the origin of species)
```

```
r ## Answers ## * 'country', 'climate', 'temperature', and 'northern_hemisphere' are ##
factors; 'has_kangaroo' is numeric. ## * using 'stringsAsFactors=FALSE' would have made
them character instead of ## factors ## * removing the quotes in temperature, northern_hemisphere,
and replacing 1 ## by TRUE in the 'has_kangaroo' column would probably what was originally
## intended.
```

The automatic conversion of data type is sometimes a blessing, sometimes an annoyance. Be aware that it exists, learn the rules, and double check that data you import in R are of the correct type within your data frame. If not, use it to your advantage to detect mistakes that might have been introduced during data entry (a letter in a column that should only contain numbers for instance.).

Inspecting `data.frame` Objects

We already saw how the functions `head()` and `str()` can be useful to check the content and the structure of a `data.frame`. Here is a non-exhaustive list of functions to get a sense of the content/structure of the data.

- Size:
 - `dim()` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the `__dim__`ensions of the object)
 - `nrow()` - returns the number of rows
 - `ncol()` - returns the number of columns

- Content:
 - `head()` - shows the first 6 rows
 - `tail()` - shows the last 6 rows
- Names:
 - `names()` - returns the column names (synonym of `colnames()` for `data.frame` objects)
 - `rownames()` - returns the row names
- Summary:
 - `str()` - structure of the object and information about the class, length and content of each column
 - `summary()` - summary statistics for each column

Note: most of these functions are “generic”, they can be used on other types of objects besides `data.frame`.

Indexing, Sequences, and Subsetting

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[2]
```

```
#> [1] "rat"
```

```
animals[c(3, 2)]
```

```
#> [1] "dog" "rat"
```

```
animals[2:4]
```

```
#> [1] "rat" "dog" "cat"
```

```
more_animals <- animals[c(1:3, 2:4)]
more_animals
```

```
#> [1] "mouse" "rat" "dog" "rat" "dog" "cat"
```

R indexes start at 1. Programming languages like Fortran, MATLAB, and R start counting at 1, because that’s what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that’s simpler for computers to do.

`:` is a special function that creates numeric vectors of integers in increasing or decreasing order, test `1:10` and `10:1` for instance. The function `seq()` (for sequence) can be used to create more complex patterns:

```
seq(1, 10, by=2)
```

```
#> [1] 1 3 5 7 9
```

```
seq(5, 10, length.out=3)
```

```
#> [1] 5.0 7.5 10.0
```

```
seq(50, by=5, length.out=10)
```

```
#> [1] 50 55 60 65 70 75 80 85 90 95
```

```
seq(1, 8, by=3) # sequence stops to stay below upper limit
```

```
#> [1] 1 4 7
```

Our temperature data frame has rows and columns (it has 2 dimensions), if we want to extract some specific data from it, we need to specify the “coordinates” we want from it. Row numbers come first, followed by column numbers.

```
temperature[1]           # first column in the data frame
temperature[1, 1]        # first element in the first column of the data frame
temperature[1, 6]        # first element in the 6th column
temperature[1:3, 7]      # first three elements in the 7th column
temperature[3, ]         # the 3rd element for all columns
temperature[, 8]         # the entire 8th column
head_temperature <- temperature[1:6, ] # temperature[1:6, ] is equivalent to head(temperature)
```

As well as using numeric values to subset a `data.frame` (or `matrix`), columns can be called by name, using one the three following notations:

```
temperature[, "City"]
temperature[["City"]]
temperature$City
```

For our purposes, these three notations are equivalent. However, the last one with the `$` does partial matching on the name. So you could also select the column "month" by typing `temperature$m`. It's a shortcut, as with all shortcuts, they can have dangerous consequences, and are best avoided. Besides, with auto-completion in RStudio, you rarely have to type more than a few characters to get the full and correct column name.

Challenge

1. The function `nrow()` on a `data.frame` returns the number of rows. Use it, in conjunction with `seq()` to create a new `data.frame` called `surveys_by_10` that includes every 10th row of the survey data frame starting at row 10 (10, 20, 30, ...)

```
### The function `nrow()` on a `data.frame` returns the number of
### rows. Use it, in conjunction with `seq()` to create a new
### `data.frame` called `temperature_by_10` that includes every 10th row
### of the temperature data frame starting at row 10 (10, 20, 30, ...)
```

```
## Answer
temperature_by_10 <- temperature[seq(10, nrow(temperature), by=10), ]
```

Conditional subsetting

Besides using the index position of an element in a vector to extract its value as we saw earlier, we can also use logical vectors:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[c(TRUE, FALSE, TRUE, TRUE)]
```

But typically, those logical vectors are not typed by hand but the result of a logical test:

```
animals != "rat"
animals[animals != "rat"]
animals[animals == "cat"]
```

If you can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions if true, OR):

```
animals[animals == "cat" & animals == "rat"] # returns nothing
animals[animals == "cat" | animals == "rat"] # returns both rat and cat
```

If you are trying to combine many conditions, it can become tedious to type. The function `%in%` allows you to test if a value is found in a vector:

```
animals %in% c("rat", "cat", "dog", "duck")
animals[animals %in% c("rat", "cat", "dog", "duck")]
```

In addition to testing equalities, you can also test whether the elements of your vector are less than or greater than a given value:

```
dates <- c(1960, 1963, 1974, 2015, 2016)
dates >= 1974
dates[dates >= 1974]
dates[dates > 1970 & dates <= 2015]
dates[dates < 1975 | dates > 2016]
```

Challenge

- Can you figure out why `"four" > "five"` returns `TRUE`?

```
# * Can you figure out why `four` > `five` returns `TRUE`?
```

```
## Answers
## * When using ">" or "<" on strings, R compares their alphabetical order. Here
## "four" comes after "five", and therefore is "greater than" it.
```