Lesson notes | Data classes and structures

Created by the GRAPH Courses team

January 2023

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

Intro
Learning objectives
Packages
What is a data class?
Numeric
Character
Quotation marks
Logical
Logical values and relational operators
Date
Converting to the YYYY-MM-DD format with {lubridate}
Introducing vectors
Creating vectors
Manipulating vectors
A common mistake: missing vector notation
Shorthand functions for numerical vectors
From vectors to data frames
Tibbles
`read_csv()` creates tibbles
Factors
Wrap-up

Intro

So far, we have focused on some of the important tools for working with data in R: the RStudio IDE, Rstudio projects and R Markdown. In this lesson, we will start to take a closer look at the ways that R stores data.

This lesson introduces many new concepts. Make sure you type along with the tutorial, so that you can develop a strong recall of these. Open a script in RStudio and type each code section out yourself.

Learning objectives

1. You can identify and create objects of the following classes: numeric, character, logical, date, and factor.

2. You can use the `class()` and `is.xxx()` family of functions (e.g. `is.numeric()` to check data classes.

3. You can use the `as.xxx()` family of functions (e.g. `as.character()`) to convert between data classes.

4. You know what relational operators (e.g. ==) are and can combine relational conditions with `&` and `|`
5. You can use the `ymd()`-like functions from {lubridate} to parse date data.
6. You can create vectors with the `c()` function.
7. You can combine vectors into data frames.
8. You understand the difference between a tibble and a data frame.

Packages

Please load the packages needed for this lesson with the code below:

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, lubridate)
```

What is a data class?

A data class is a way of categorizing data based on the type of values that it can take. In R, there are five main data classes that you need to be aware of: numeric, character, logical, date and factor. Each is described in more detail below.

Numeric

Let's start with the numeric class, which is used for data that contains, well, numbers. This could be an integer or a decimal, like 25 or 23.4.

You can verify the class of numbers (or any other data type) with the built-in `class()` function:

```
class(4)
```

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

```
class(0.1)
```

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

The function `is.numeric()` is also used to verify that an object is numeric:

```
is.numeric(4)
```

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

```
is.numeric("Bob") # Not numeric
```

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

(Apart from `is.numeric()`, there is a whole family of other `is.XXX` functions, such as `is.character()`. You will see these below.)

Numeric data can sometimes be represented with scientific notation, where "e" refers to "10 to the power of". For example, we can write the number 2000 as 2 times 10 to the power of 3, `2e3`:

```
2e3
```

```
## [1] 2000
```

"Integers" (numbers without any decimal) are a special class of numbers, represented with an "L" after the number:

```
4L
```

```
## [1] 4
```

```
class(4L)
```

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

However, you usually should not have to use the `L` notation; to write a whole number like 4, just write `4`, not `4L`.

You may also see the terms "real" or "double" (abbreviated as "dbl") used to describe numeric data. The differences between these are only relevant for advanced users. You can ignore them for now.

Each line of code below tries to define a numeric object (the numbers 1 to 4). But all lines have a mistake, and do not properly define the object. Try to find the four mistakes, fix them and perform the assignment:

```
numeric_obj1 <- `1L` # define the number 1
numeric_obj2 <-  two # define the number 2
numeric_obj3 <- "3"  # define the number 3
numeric_obj4 <-  4,0 # define the number 4.0
numeric_obj5 <-  5E0 # define the number 5
```

Character

The character class is used for data that contains text. In R, we write text by putting it in quotation marks, like this:

```
"A piece of text."
```

We can check this object's class like so:

```
class("A piece of text.")
```

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

```
is.character("A piece of text.")
```

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

Note that if you wrap a number in quotes, it automatically becomes a character, according to R:

```
class(4) # numeric
```

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

```
class("4") # Now a character
```

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

Character values are sometimes referred to as "strings" or "character strings". You can use these terms interchangeably.

Quotation marks

You can use either single, `'`, or double quotation marks, `"`, to create a character string. The tidyverse style guide, which we use, recommends that you use double quotes in most cases.

Notably, if you start a string with a single quote, you must also close it with a single quote; the same goes for double quotes. So a string like `"Hello World'` is not properly defined and will cause an error, because it opens with double quotes, but closes with a single quote.

If quotation marks are used to define character strings, what should you do when your string already has a quotation mark within it?

For example, how would you create the following string `Obi said "Hello World"`.

Here, you have two options: you can double quotes internally, with single quotes to wrap the whole string:

```
obi_string <- 'Obi says "Hello World"'
obi_string
```

```
## [1] "Obi says \"Hello World\""
```

or you can use single quotes internally, with double quotes to wrap the whole string:

```
obi_string_2 <- "Obi says 'Hello World'"
obi_string_2
```

```
## [1] "Obi says 'Hello World'"
```

If you try to use double quotes both for the internal quote, and externally to wrap the string, you will get an error:

```
obi_string_3 <- "Obi says "Hello World""
```

```
Error: object 'obi_string_3' not found
```

The same thing will happen if you use single quotes internally and externally.

Each line of code below tries to define a character object. But all lines EXCEPT ONE have a mistake, and do not properly define the object. Try to find the four mistakes, fix them and perform the assignment:

```
char_object1 <-  Hello World
char_object2 <- 'My name is Ike'
char_object3 <- 'I am 24' years old
char_object4 <- 'I'm learning R"
char_object5 <- `for data science`
```

Note that you cannot perform math operations on character values. For example, the code below gives an error:

```
sqrt("100") # square root. Does not work
```

```
Error in sqrt("100") : non-numeric argument to mathematical
function
```

But we can convert that character into a numeric class with the function `as.numeric()`, and then the function will run:

```
sqrt(as.numeric("100"))
```

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

The `as.numeric()` function seen above is part of a family of functions for converting between classes. There are many others. We could for example, convert a number into a character:

```
4
```

```
## [1] 4
```

```
as.character(4)
```

```
## [1] "4"
```

Above, you can observe that the `as.character(4)` line prints its output with quotes, indicating that it is a character.

Logical

Logical data contains two values, `TRUE` or `FALSE`, which are written in all capital letters. These can also be written in short as `T` and `F`. Logical data can be thought of like a light switch: it's either on (`TRUE`) or off (`FALSE`).

Logical values are often used as the arguments to functions, for example:

```
mean(airquality$Ozone)
```

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

```
mean(airquality$Ozone, na.rm = TRUE) # remove NAs to calculate mean
```

```
## [1] 42.12931
```

The second code line, with `na.rm = TRUE` is an example of the use of the logical `TRUE` value as the argument to a function.

Each line of code below tries to define a logical object. But all lines, EXCEPT ONE, have a mistake, and do not properly define the object. Try to find the four mistakes, fix them and perform the assignment.

```
logical_obj1 <- f
logical_obj2 <- False
logical_obj3 <- "True"
logical_obj4 <- F
logical_obj5 <- `TRUE`
```

Logical values and relational operators

Logical values are returned when you apply *relational operators* in R.

A relational operator (sometimes called comparison operators) tests the relationship between two values. You will consider them in detail in a future lesson, but here we'll give a few examples:

The greater-than, >, operator checks whether the left-hand-side (LHS) object is greater than the right-hand-side (RHS) object:

```
3 > 4 # is 3 greater than 4? Answer is FALSE
```

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

The == comparator checks whether two values are equal:

```
3 == 3 # is 3 equal to 3? Answer is TRUE. Note the double equals sign here.
```

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

The <= operator checks whether the LHS is less than or equal to the RHS object:

```
3 <= 3 # is 3 less than or equal to 3? Answer is TRUE
```

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

Logical values can be combined using the ampersand, "&", which means "AND", or the vertical bar, "|", which means "OR".

& checks whether ALL values are true:

```
TRUE & TRUE # All values are true. Returns TRUE
```

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

```
TRUE & FALSE # ALL values are not true. Returns FALSE
```

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

| checks whether AT LEAST ONE value is true:

```
TRUE | FALSE # At least one value is true. Returns TRUE
```

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

```
FALSE | FALSE # No value is true. Returns FALSE
```

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

It will become clearer how `&` and `|` work when you use these in the context of real datasets. So if it feels unclear now, feel free to ignore it.

Predict whether each line below will evaluate to TRUE or FALSE. Then use your R code console to check these.

```
is.numeric(5)
5 == 5
5 > 6
is.numeric(5) | 5 > 6
is.numeric(5) & 5 > 6
```

Date

The `Date` class contains dates, which must be formatted in YYYY-MM-DD format (e.g. `"2022-12-31"`), with four digits for the year, two digits for the month, and two digits for the day of the month.

Of course if you just put in such a date string, R will initially consider this to be a character:

```
class("2022-12-31")
```

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

In order for R to recognize a data value as a date, you use the `as.Date()` function:

```
my_date <- as.Date("2022-12-31")
class(my_date)
```

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

WATCH OUT

Note the capital "D" in the `as.Date()` function!

With the date format, you can now do things like find the difference between two dates:

```r
as.Date("2022-12-31") - as.Date("2022-12-20")
```

```
## Time difference of 11 days
```

This would of course not be possible if you had bare characters:

```r
"2022-12-31" - "2022-12-20"
```

```
Error in "2022-12-31" - "2022-12-20" :
  non-numeric argument to binary operator
```

Note that if you use any other date format than YYYY-MM-DD, R's `as.Date()` function will not work:

```r
as.Date("12/31/2022") # Common America date format MM/DD/YYYY
as.Date("Dec 31, 2022") # Common America date format MM/DD/YYYY
```

```
Error in charToDate(x) :
  character string is not in a standard unambiguous format
```

Each line of code below tries to define a date object. But all lines, EXCEPT ONE, have a mistake, and do not properly define the object. Try to find the four mistakes, fix them and perform the assignment.

```r
date_obj1 <- as.date("2022-12-31")
date_obj2 <- as.date(2022-12-30)
date_obj3 <- as.Date("2022-12-29")
date_obj4 <- as.Date("12/28/2022")
date_obj5 <- "2022-12-27"
```

Converting to the YYYY-MM-DD format with {lubridate}

Because R only recognizes the "YYYY-MM-DD" format as a date, you will often have to convert from other date formats into this format. The {lubridate} package makes this very easy. It has a family of intelligent date-parsing functions, which are named in terms of the relative arrangement of year, month and date.

So you have the `mdy()` function (which stands for month, day, year), `dmy()` (day, month, year), `ymd()` (year, month, day), and so on.

Run the lines of code below to observe how these `lubridate` date parsing functions work.

```r
mdy("December 31 2001")
```

```
## [1] "2001-12-31"
```

```
mdy("Dec 31 2001")
```

```
## [1] "2001-12-31"
```

```
mdy("Dec-31-01")
```

```
## [1] "2001-12-31"
```

```
mdy("01/31/2001")
```

```
## [1] "2001-01-31"
```

```
ymd("2001 December 31st")
```

```
## [1] "2001-12-31"
```

```
ymd("2001-Dec-31")
```

```
## [1] "2001-12-31"
```

Do you see the beauty of this? You do not need to worry about whether a hyphen or a slash or a space was used to separate the dates, or whether the months were spelled out in full or abbreviated. All you need to know is the intended order of the date components (day, month and year) and voila!

Note that the lubridate functions automatically do the `as.Date()` conversion to convert the values from characters to dates:

```
class("December 31 2001") # recognized only as a character
```

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

```
class(mdy("December 31 2001")) # after applying lubridate, R recognizes value
        as date
```

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

Convert the following to R dates with the `ymd` family of functions from lubridate:

```
"March 29, 2022"
"2022 March 29"
"2022 Mar 29"
"2022/03/29"
```

R stores dates internally as the number of days since January 1, 1970. This means that the date `January 1, 1970` is represented as `0`, while `January 2, 1970` is represented as `1`, and so on. You can see this by running converting those dates to numbers:

```
as.numeric(as.Date("1970-01-01"))
```

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

```
as.numeric(as.Date("1970-01-02"))
```

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

```
as.numeric(as.Date("2022-12-31"))
```

```
## [1] 19357
```

This information is sometimes useful when importing date data.

Introducing vectors

So far, we have been looking at data that contains just a single value. But of course, most data comes in some kind of collection—a data *structure*. The data structure you are most familiar with is a data table (sometimes called a spreadsheet, but typically represented as a data frame in R).

But the most fundamental data structures in R are actually *vectors*. Let's spend some time thinking about these.

A vector is a collection of values that all have the same class (for example, all numeric or all character). It may be helpful to think of a vector as a column or row in an Excel spreadsheet.

Creating vectors

Vectors can be created using the `c()` function, with the components of the vector separated by commas. For example, the code `c(1, 2, 3)` defines a vector with the elements `1`, `2` and `3`.

In your script, define the following vectors:

```
my_numeric_vec <- c(0, 1, 1, 2, 3)
my_numeric_vec # print the vector
```

```
## [1] 0 1 1 2 3
```

```
my_numeric_vec2 <- c(4, 5, 3, 4, 1)
my_numeric_vec2 # print the vector
```

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

```
my_character_vec <- c("Bob", "Jane", "Joe", "Obi", "Aka")
my_character_vec # print the vector
```

```
## [1] "Bob"  "Jane" "Joe"  "Obi"  "Aka"
```

```
my_logical_vec <- c(T, T, F, F, F)
my_logical_vec # print the vector
```

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

You can also check the classes of these vectors:

```
class(my_numeric_vec)
```

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

```
class(my_character_vec)
```

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

```
class(my_logical_vec)
```

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

Each line of code below tries to define a vector with three elements. But all lines, EXCEPT ONE, have a mistake, and do not properly define the vector object. Try to find the four mistakes, fix them and perform the assignment.

```
my_vec_1 <- (1,2,3)
my_vec_2 <- 2,3,4
my_vec_3 <- "c(4,5,6)"
my_vec_4 <- c("Obi", "Chika" "Nonso")
my_vec_5 <- as.Date(c("2020-10-10", "2020-10-11", "2020-10-13"))
```

The individual values within a vector are called *components* or elements. So the vector `c(1, 2, 3)` has three components/elements.

Manipulating vectors

Many of the functions and operations you have encountered so far in the course can be applied to vectors.

For example, we can multiply our `my_numeric_vec` object by 2:

```
my_numeric_vec
```

```
## [1] 0 1 1 2 3
```

```
my_numeric_vec * 2
```

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

Notice that every element in the vector was multiplied by 2.

Or, below we take the square root of `my_numeric_vec2`:

```
my_numeric_vec2
```

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

```
sqrt(my_numeric_vec2)
```

```
## [1] 2.000000 2.236068 1.732051 2.000000 1.000000
```

You can also can add (numeric) vectors to each other:

```
my_numeric_vec + my_numeric_vec2
```

```
## [1] 4 6 4 6 4
```

Note that the first element of `my_numeric_vec` is added to the first element of `my_numeric_vec2` and the second element of `my_numeric_vec` is added to the second element of `my_numeric_vec2` and so on.

Below are some other functions you could run on vector. Type them out in your console and observe their outputs:

```
head(my_character_vec, 2) # first two elements
table(my_logical_vec)
length(my_logical_vec)
sort(my_numeric_vec2)
sort(my_character_vec) # sorts in alphabetical order
```

Consider the vector defined here, which holds the hand circumference, in inches, of children:

```
circum_inches <- c(4.0, 5.6, 3.4, 5.8, 5.6, 4.0)
```

Each line of code below tries to run a function on this vector. But all lines, EXCEPT ONE, have a mistake, and do not properly carry out the function. Try to find the four mistakes and fix them.

```
sum()circum_inches # find the sum of the vector
head(3, circum_inches) # take the first three elements of the vector
sort(circum_inches, decreasing = false) # sort the vector in increasing order
Table(circum_inches)  # frequency table
janitor::tabyl(circum_inches)  # frequency table
```

```
## Error: <text>:1:6: unexpected symbol
## 1: sum()circum_inches
##          ^
```

A common mistake: missing vector notation

An error that students frequently encounter involves failing to create a vector where one is needed. For example, consider the code line below:

```
mean(1,2,3,4)
```

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

Hmmm. The mean of 1, 2, 3 and 4 is definitely not 1. What is going on. This is happening because the primary argument to `mean()` must be a vector:

```
mean(c(1,2,3,4))
```

```
## [1] 2.5
```

Now all good!

Watch out for this/

Using the `median()` function, find the median of this collection of numbers: `1, 5, 2, 8, 9, 10, 11, 46, 23, 45, 2, 4, 5, 6`. (Remember to put the number collection in a vector!)

Shorthand functions for numerical vectors

R has a number of shorthand functions for creating numerical vectors. The most commonly used of these is the colon operator, `:`, which creates a sequence of integers:

```
1:5 # 1 to 5
```

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

```r
100:113 # 100 to 113
```

```
##  [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113
```

You can also use the `seq()` function to create a sequence of numbers as a vector:

For example, to create a sequence of numbers from 1 to 10, you can run:

```r
seq(from = 1, to = 10)
```

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

By default, the increment is set to 1. To use a different increment, just change the value of the `by` argument. For example, to create a sequence with an increment of two:

```r
seq(from = 1, to = 10, by = 2)
```

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

It's also possible to create descending sequences by using `:` or `seq()`:

```r
10:1
```

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

```r
seq(from = 10, to = 1, by = -2)
```

```
## [1] 10  8  6  4  2
```

Although we defined a vector as a *collection* of objects, in R even single values are technically considered vectors! You can check this with the `is.vector()` function:

```r
is.vector(c(1,2,3)) # obviously a vector
```

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


```
is.vector(1) # still a vector!
```

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

From vectors to data frames

Now that we have a handle on creating vectors, let's move on to the most commonly used object in R: data frames. A data frame is just a collection of vectors of the same length with some helpful metadata. We can create one using the `data.frame()` function.

In the below example, we first create vector variables (age, sex and comorbidity) for six individuals:

```
age <- c(18, 25, 46, 54, 60, 72)
sex <- c('M', 'F', 'F', 'M', 'M', 'F')
comorbidity <- c(T, T, F, F, F, T)
```

We can now use the `data.frame()` function to combine these into a single tabular structure:

```
data_epi <- data.frame(age, sex, comorbidity)
data_epi
```

```
##   age sex comorbidity
## 1  18   M        TRUE
## 2  25   F        TRUE
## 3  46   F       FALSE
## 4  54   M       FALSE
## 5  60   M       FALSE
## 6  72   F        TRUE
```

Note that instead of creating each vector separately, you can create your data frame defining each of the vectors inside the `data.frame()` function.

```
data_epi <- data.frame(age = c(18, 25, 46, 54, 60, 72),
                       sex = c('M', 'F', 'F', 'M', 'M', 'F'),
                       comorbidity = c(T ,T ,F ,F ,F ,T))

data_epi
```

```
##   age sex comorbidity
## 1  18   M        TRUE
```

```
## 2   25    F         TRUE
## 3   46    F        FALSE
## 4   54    M        FALSE
## 5   60    M        FALSE
## 6   72    F         TRUE
```

We can check the class of this data frame:

```
class(data_epi)
```

```
## [1] "data.frame"
```

Most of the time you work with data in R, you will be importing it from external contexts. But it is sometimes useful to create datasets *within* R itself. It is in such cases that the `data.frame()` function will come in handy.

To create a data frame from vectors, all the vectors **must** have the same length. Otherwise you will get an error. For example:

```
mydf <- data.frame(age = c(5, 9, 8),
                   sex = c('M', 'F'))
```

```
Error in data.frame(age = c(5, 9, 8), sex = c("M", "F")) :
  arguments imply differing number of rows: 3, 2
```

To extract the vectors back out of the data frame, use the `$` syntax. Run the following lines of code in your console to observe this.

```
data_epi$age
is.vector(data_epi$age) # verify that this column is indeed a vector
class(data_epi$age) # check the class of the vector
```

Earlier, we defined a data frame with the following vectors below. Combine these into a data frame, with the following column names: "number_of_children" for the numeric vector, "name" for the character vector and "is_married" for the logical vector.

```
my_numeric_vec <- c(0, 1, 1, 2, 3)
my_character_vec <- c("Bob", "Jane", "Joe", "Obi", "Aka")
my_logical_vec <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
```

Use the `data.frame()` function to define a data frame in R that resembles the following table:

room	number_of_windows
dining	3
kitchen	2
bedroom	5

Tibbles

The default version of tabular data in R is called a data frame, but there is another representation of tabular data provided by the *tidyverse* package. It's called a `tibble`, and it is an improved version of `data.frame`.

You can create a tibble using the `tibble()` function. (Remember to import the tidyverse package to use its functions.)

```
tibble_epi <- tibble(
          age = c(18, 25, 46, 54, 60, 72),
          sex = c('M', 'F', 'F', 'M', 'M', 'F'),
          comorbidity = c(T,T,F,F,F,T)
          )
tibble_epi
```

```
## # A tibble: 6 × 3
##     age sex   comorbidity
##   <dbl> <chr> <lgl>
## 1    18 M     TRUE
## 2    25 F     TRUE
## 3    46 F     FALSE
## 4    54 M     FALSE
## 5    60 M     FALSE
## 6    72 F     TRUE
```

Notice that the tibble gives the data dimensions in the first line:

```
👉# A tibble: 6 × 3👈
    age sex   comorbidity
  <dbl> <chr> <lgl>
1    18 M     TRUE
2    25 F     TRUE
```

And also tells you the data types, at the top of each column:

```
# A tibble: 6 × 3
    age sex   comorbidity
  <dbl> <chr> <lgl>
1    18 M     TRUE
2    25 F     TRUE
```

There, "dbl" stands for double (which is a kind of numeric class), "chr" stands for character, and "lgl" for logical.

You can convert a `data.frame` to `tibble` using the `as_tibble` function:

```
df <- data.frame(
          age = c(18, 25, 46, 54, 60, 72),
          sex = c('M', 'F', 'F', 'M', 'M', 'F'),
          comorbidity = c(T, T, F, F, F, T)
          )

a_tibble <-  as_tibble(df)

a_tibble
```

```
## # A tibble: 6 × 3
##     age sex   comorbidity
##   <dbl> <chr> <lgl>
## 1    18 M     TRUE
## 2    25 F     TRUE
## 3    46 F     FALSE
## 4    54 M     FALSE
## 5    60 M     FALSE
## 6    72 F     TRUE
```

And you can convert a tibble back to a data frame with `as.data.frame()`:

```
as.data.frame(a_tibble)
```

```
##   age sex comorbidity
## 1  18   M        TRUE
## 2  25   F        TRUE
## 3  46   F       FALSE
## 4  54   M       FALSE
## 5  60   M       FALSE
## 6  72   F        TRUE
```

For your most of your data analysis needs, you should prefer tibbles over regular data frames.

`read_csv()` creates tibbles

When you import data with the `read_csv()` function from {readr}, you get a tibble:

```
ebola_tib <- read_csv("https://tinyurl.com/ebola-data-sample") # Needs
        internet to run
class(ebola_tib)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

But when you import data with the base `read.csv()` function, you get a data.frame:

```
ebola_df <- read.csv("https://tinyurl.com/ebola-data-sample") # Needs internet
        to run
class(ebola_df)
```

```
## [1] "data.frame"
```

Try printing `ebola_tib` and `ebola_df` to your console to observe the different printing behavior of tibbles and data frames.

The `iris` data frame is one of R's built-in datasets. Convert it to a tibble with the `as_tibble()` function. Then print it to your console. How does the tibble output differ from the original `iris` data frame?

Factors

Finally, let's turn briefly to factors, which is another data class. We left this class until the end because understanding factors requires that you understand vectors.

A factor is a nominal (categorical) variable with a set of known possible values called levels.

Why might we use a factor class? The most common reason are:

- to force characters to sort in a custom order
- to show zero counts

What these mean will become clear by considering an example.

Imagine that you have a variable that records the month of birth for a number of infants:

```
birth_month <- c("Dec", "Apr", "Jan", "Mar", "Oct", "Nov", "Jan", "Apr")
```

And you want to count the number of births per month. You could use the base `table()` function:

```
table(birth_month)
```

```
## birth_month
## Apr Dec Jan Mar Nov Oct
##   2   1   2   1   1   1
```

We see that 2 babies were born in April, 1 in December, and so on.

You could also use the `tabyl()` function from {janitor} for this:

```
tabyl(birth_month)
```

```
##  birth_month n percent
##          Apr 2   0.250
##          Dec 1   0.125
##          Jan 2   0.250
##          Mar 1   0.125
##          Nov 1   0.125
##          Oct 1   0.125
```

But do you see a problem with these outputs? The months are sorted in alphabetical order(!) Indeed if you try to `sort()` the `birth_month` vector directly, you will note the same thing:

```
sort(birth_month)
```

```
## [1] "Apr" "Apr" "Dec" "Jan" "Jan" "Mar" "Nov" "Oct"
```

But this is not a sensible way to sort this variable. A chronological order, with January first, would be much better.

You can fix this problem with a factor. To create a factor you use the `factor()` function, with your original character vector and a list of valid **levels**, arranged in the correct order:

```
birth_month_factor <- factor(x = birth_month,
                             levels = c("Jan", "Feb", "Mar", "Apr",
                                        "May", "Jun", "Jul", "Aug",
                                        "Sep", "Oct", "Nov", "Dec"))

class(birth_month_factor) # check its class
```

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

```
birth_month_factor # print it
```

```
## [1] Dec Apr Jan Mar Oct Nov Jan Apr
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Notice that the levels are listed in the output.

```
[1] Dec Apr Jan Mar Oct Nov Jan Apr
👉Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec👈
```

Now, we can sort the vector properly:

```
sort(birth_month_factor)
```

```
## [1] Jan Jan Mar Apr Apr Oct Nov Dec
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

And if we create the frequency count tables, we get the right order:

```
table(birth_month_factor)
```

```
## birth_month_factor
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
##   2   0   1   2   0   0   0   0   0   1   1   1
```

```
tabyl(birth_month_factor)
```

```
##  birth_month_factor n percent
##                 Jan 2   0.250
##                 Feb 0   0.000
##                 Mar 1   0.125
##                 Apr 2   0.250
##                 May 0   0.000
##                 Jun 0   0.000
##                 Jul 0   0.000
##                 Aug 0   0.000
##                 Sep 0   0.000
##                 Oct 1   0.125
##                 Nov 1   0.125
##                 Dec 1   0.125
```

As you can see, months with zero counts are also included in the table outputs. This will often be useful!

If you would rather not see these zero count months, the `tabyl()` function allows you to drop them, by using the `show_missing_levels` argument:

```
tabyl(birth_month_factor, show_missing_levels = FALSE)
```

```
##  birth_month_factor n percent
##                 Jan 2   0.250
##                 Mar 1   0.125
##                 Apr 2   0.250
##                 Oct 1   0.125
##                 Nov 1   0.125
##                 Dec 1   0.125
```

The variable `visit_day` below records the day of the week that a clinic was visited.

```
visit_day <- c("Mon", "Mon", "Tue", "Fri", "Thu", "Wed", "Sun", "Sat", "Tue")
```

Convert this into a factor with the `factor()` function. The levels should be in order of the days of the week, starting with "Sun" and ending with "Sat".

Then create a frequency table of this variable using the `tabyl()` function. Does the table sort in the proper chronological order?

Wrap-up

You've learned a lot in this lesson! You now know about all of the basic R data classes (numeric, character, logical, date, factor) and how to create objects of each class. You also know how to check an object's class with `class()` and convert between classes with `as.xxx()`. Finally, you know how to create vectors and data frames.

With this knowledge, you are now ready to start doing some serious data analysis in R. In the coming lessons, you'll start to learn about the dplyr package, which will provide you with powerful tools for manipulating your data frames and tibbles.

Congratulations on making it this far! You have covered a lot and should be proud of yourself.

Contributors

The following team members contributed to this lesson:

DANIEL CAMARA

Data Scientist at the GRAPH Network and fellowship as Public Health researcher at Fiocruz, Brazil

Passionate about lots of things, especially when it involves people leading lives with more equality and freedom

EDUARDO ARAUJO

Student at Universidade Tecnologica Federal do Parana
Passionate about reproducible science and education

LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

Some material in this lesson was adapted from the following sources:

- Wickham, H., & Grolemund, G. (n.d.). *R for data science*. 15 Factors | R for Data Science. Accessed October 26, 2022. https://r4ds.had.co.nz/factors.html.