February 3, 2023

The results below are generated from an R script.

````
---
title: "Starting with data"
author: "Data Carpentry contributors"
minutes: 20
---

```{r, echo=FALSE, purl=FALSE, message = FALSE}
source("setup.R")
```

## Error:  attempt to use zero-length variable name
````

```
> ### Learning Objectives
>
> * Load external data from a .csv file into a data frame.
> * Install and load packages.
> * Describe what a data frame is.
> * Summarize the contents of a data frame.
> * Use indexing to subset specific portions of data frames.
> * Describe what a factor is.
> * Convert between strings and factors.
> * Reorder and rename factors.
> * Change how character strings are handled in a data frame.
> * Format dates.

## Error:  <text>:2:1:  unexpected '>'
## 1:
## 2:  >
##    ^
```

````
## Loading the survey data

```{r, echo=FALSE, purl=TRUE}
### Loading the survey data
```

We are investigating the animal species diversity and weights found within plots
at our study site. The dataset is stored as a comma separated value (CSV) file.
Each row holds information for a single animal, and the columns represent:

| Column           | Description                                |
````

```
|------------------|-------------------------------------------|
| record\_id       | Unique id for the observation             |
| month            | month of observation                      |
| day              | day of observation                        |
| year             | year of observation                       |
| plot\_id         | ID of a particular experimental plot of land |
| species\_id      | 2-letter code                             |
| sex              | sex of animal ("M", "F")                  |
| hindfoot\_length | length of the hindfoot in mm              |
| weight           | weight of the animal in grams             |
| genus            | genus of animal                           |
| species          | species of animal                         |
| taxon            | e.g. Rodent, Reptile, Bird, Rabbit        |
| plot\_type       | type of plot                              |

### Downloading the data

We created the folder that will store the downloaded data (‘data_raw‘) in the chapter
["Before we start"](https://datacarpentry.org/R-ecology-lesson/00-before-we-start.html#Organizing_your_w
If you skipped that part, it may be a good idea to have a look now, to make
sure your working directory is set up properly.

We are going to use the R function ‘download.file()‘ to download the CSV file
that contains the survey data from Figshare, and we will use ‘read_csv()‘ to
load the content of the CSV file into R.

Inside the ‘download.file‘ command, the first entry is a character string with the
source URL ("https://ndownloader.figshare.com/files/2292169").
This source URL downloads a CSV file from figshare. The text after the comma
("data_raw/portal_data_joined.csv") is the destination of the file on your local
machine. You'll need to have a folder on your machine called "data_raw" where
you'll download the file. So this command downloads a file from Figshare, names
it "portal_data_joined.csv" and adds it to a preexisting folder named "data_raw".

‘‘‘{r, eval=FALSE, purl=TRUE}
download.file(url = "https://ndownloader.figshare.com/files/2292169",
              destfile = "data_raw/portal_data_joined.csv")
‘‘‘


### Reading the data into R

The file has now been downloaded to the destination you specified, but R has not
yet loaded the data from the file into memory. To do this, we can use the
‘read_csv()‘ function from the **‘tidyverse‘** package.

Packages in R are basically sets of additional functions that let you do more
stuff. The functions we've been using so far, like ‘round()‘, ‘sqrt()‘, or ‘c()‘
come built into R. Packages give you access to additional functions beyond base R.
A similar function to ‘read_csv()‘ from the tidyverse package is ‘read.csv()‘ from
base R. We don't have time to cover their differences but notice that the exact
spelling determines which function is used.
Before you use a package for the first time you need to install it on your
```

```
machine, and then you should import it in every subsequent R session when you
need it.

To install the **`tidyverse`** package, we can type
`install.packages("tidyverse")` straight into the console. In fact, it's better
to write this in the console than in our script for any package, as there's no
need to re-install packages every time we run the script.
Then, to load the package type:

```{r, message = FALSE, purl = FALSE}
## load the tidyverse packages, incl. dplyr
library(tidyverse)
```

Now we can use the functions from the **`tidyverse`** package.
Let's use `read_csv()` to read the data into a data frame
(we will learn more about data frames later):

```{r, eval=TRUE,  purl=FALSE}
surveys <- read_csv("data_raw/portal_data_joined.csv")
```

When you execute `read_csv` on a data file, it looks through the first 1000 rows
of each column and guesses its data type. For example, in this dataset,
`read_csv()` reads `weight` as `col_double` (a numeric data type), and `species`
as `col_character`. You have the option to specify the data type for a column
manually by using the `col_types` argument in `read_csv`.

> ### Note
>
> `read_csv()` assumes that fields are delineated by commas. However, in several
> countries, the comma is used as a decimal separator and the semicolon (;) is
> used as a field delineator. If you want to read in this type of files in R,
> you can use the `read_csv2()` function. It behaves like `read_csv()` but
> uses different parameters for the decimal and the field separators.
There is also the `read_tsv()` for tab separated data files and `read_delim()`
> for less common formats.
> Check out the help for `read_csv()` by typing `?read_csv` to learn more.
>
> In addition to the above versions of the csv format, you should develop the habits
> of looking at and recording some parameters of your csv files. For instance,
> the character encoding, control characters used for line ending, date format
> (if the date is not split into three variables), and the presence of unexpected
> [newlines](https://en.wikipedia.org/wiki/Newline) are important characteristics of your data files.
> Those parameters will ease up the import step of your data in R.

We can see the contents of the first few lines of the data by typing its
name: `surveys`. By default, this will show you as many rows and columns of
the data as fit on your screen.
If you wanted the first 50 rows, you could type `print(surveys, n = 50)`

We can also extract the first few lines of this data using the function
`head()`:
```

```{r, results='show', purl=FALSE}
head(surveys)
```

Unlike the `print()` function, `head()` returns the extracted data. You could
use it to assign the first 100 rows of `surveys` to an object using
`surveys_sample <- head(surveys, 100)`. This can be useful if you want to try
out complex computations on a subset of your data before you apply them to the
whole data set.
There is a similar function that lets you extract the last few lines of the data
set. It is called (you might have guessed it) `tail()`.

To open the dataset in RStudio's Data Viewer, use the `view()` function:

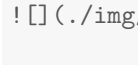
```{r, eval = FALSE, purl = FALSE}
view(surveys)
```

> ### Note
>
> There are two functions for viewing which are case-sensitive. Using `view()` with a
> lowercase 'v' is part of tidyverse, whereas using `View()` with an uppercase 'V' is
> loaded through base R in the `utils` package.

What are data frames?

When we loaded the data into R, it got stored as an object of class `tibble`,
which is a special kind of data frame (the difference is not important for our
purposes, but you can learn more about tibbles
[here](https://tibble.tidyverse.org/)).
Data frames are the _de facto_ data structure for most tabular data, and what we
use for statistics and plotting.
Data frames can be created by hand, but most commonly they are generated by
functions like `read_csv()`; in other words, when importing
spreadsheets from your hard drive or the web.

A data frame is the representation of data in the format of a table where the
columns are vectors that all have the same length. Because columns are
vectors, each column must contain a single type of data (e.g., characters, integers,
factors). For example, here is a figure depicting a data frame comprising a
numeric, a character, and a logical vector.



We can see this also when inspecting the structure of a data frame
with the function `str()`:

```{r, purl=FALSE}
str(surveys)
```

```
## Inspecting data frames

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. Let's try them out!

* Size:
    * `dim(surveys)` - 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(surveys)` - returns the number of rows
    * `ncol(surveys)` - returns the number of columns

* Content:
    * `head(surveys)` - shows the first 6 rows
    * `tail(surveys)` - shows the last 6 rows

* Names:
    * `names(surveys)` - returns the column names (synonym of `colnames()` for `data.frame`
   objects)
    * `rownames(surveys)` - returns the row names

* Summary:
    * `str(surveys)` - structure of the object and information about the class, length and
   content of  each column
    * `summary(surveys)` - summary statistics for each column

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


> ### Challenge
>
> Based on the output of `str(surveys)`, can you answer the following questions?
>
> * What is the class of the object `surveys`?
> * How many rows and how many columns are in this object?
>
> ```{r, answer=TRUE, results="markup", purl=FALSE}
>
> str(surveys)
>
> ## * class: data frame
> ## * how many rows: 34786,  how many columns: 13
>
> ```


```{r, echo=FALSE, purl=TRUE}

## Challenge
## Based on the output of `str(surveys)`, can you answer the following questions?
##
## * What is the class of the object `surveys`?
```
```

```
## * How many rows and how many columns are in this object?

```


Indexing and subsetting data frames

```{r, echo=FALSE, purl=TRUE}

## Indexing and subsetting data frames
```


Our survey 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. However, note that different ways of specifying these coordinates lead to results with different classes.

```{r, purl=FALSE}
# We can extract specific values by specifying row and column indices
# in the format:
# data_frame[row_index, column_index]
# For instance, to extract the first row and column from surveys:
surveys[1, 1]

# First row, sixth column:
surveys[1, 6]

# We can also use shortcuts to select a number of rows or columns at once
# To select all columns, leave the column index blank
# For instance, to select all columns for the first row:
surveys[1, ]

# The same shortcut works for rows --
# To select the first column across all rows:
surveys[, 1]

# An even shorter way to select first column across all rows:
surveys[1] # No comma!

# To select multiple rows or columns, use vectors!
# To select the first three rows of the 5th and 6th column
surveys[c(1, 2, 3), c(5, 6)]

# We can use the : operator to create those vectors for us:
surveys[1:3, 5:6]

# This is equivalent to head_surveys <- head(surveys)
head_surveys <- surveys[1:6, ]

# As we've seen, when working with tibbles
# subsetting with single square brackets ("[]") always returns a data frame.
```

```
# If you want a vector, use double square brackets ("[[]]")

# For instance, to get the first column as a vector:
surveys[[1]]

# To get the first value in our data frame:
surveys[[1, 1]]
```

`:` is a special function that creates numeric vectors of integers in increasing
or decreasing order, test `1:10` and `10:1` for instance.

You can also exclude certain indices of a data frame using the "`-`" sign:

```{r, purl=FALSE}
surveys[, -1]                  # The whole data frame, except the first column
surveys[-(7:nrow(surveys)), ] # Equivalent to head(surveys)
```

Data frames can be subset by calling indices (as shown previously), but also by calling their column nam

```{r, eval = FALSE, purl=FALSE}
# As before, using single brackets returns a data frame:
surveys["species_id"]
surveys[, "species_id"]

# Double brackets returns a vector:
surveys[["species_id"]]

# We can also use the $ operator with column names instead of double brackets
# This returns a vector:
surveys$species_id
```

In RStudio, you can use the autocompletion feature to get the full and correct names of the columns.

```
> ### Challenge
>
> 1. Create a `data.frame` (`surveys_200`) containing only the data in
>    row 200 of the `surveys` dataset.
>
> 2. Notice how `nrow()` gave you the number of rows in a `data.frame`?
>
>      * Use that number to pull out just that last row from the `surveys` dataset.
>      * Compare that with what you see as the last row using `tail()` to make
>        sure it's meeting expectations.
>      * Pull out that last row using `nrow()` instead of the row number.
>      * Create a new data frame (`surveys_last`) from that last row.
>
> 3. Use `nrow()` to extract the row that is in the middle of the data
>    frame. Store the content of this row in an object named `surveys_middle`.
>
> 4. Combine `nrow()` with the `-` notation above to reproduce the behavior of
```

```
>    `head(surveys)`, keeping just the first through 6th rows of the surveys
>    dataset.
>
> ```{r, answer=TRUE, purl=FALSE}
> ## 1.
> surveys_200 <- surveys[200, ]
> ## 2.
> # Saving `n_rows` to improve readability and reduce duplication
> n_rows <- nrow(surveys)
> surveys_last <- surveys[n_rows, ]
> ## 3.
> surveys_middle <- surveys[n_rows / 2, ]
> ## 4.
> surveys_head <- surveys[-(7:n_rows), ]
> ```


```{r, echo=FALSE, purl=TRUE}
### Challenges:
###
### 1. Create a `data.frame` (`surveys_200`) containing only the
###    data in row 200 of the `surveys` dataset.
###
### 2. Notice how `nrow()` gave you the number of rows in a `data.frame`?
###
###      * Use that number to pull out just that last row in the data frame
###      * Compare that with what you see as the last row using `tail()` to make
###        sure it's meeting expectations.
###      * Pull out that last row using `nrow()` instead of the row number
###      * Create a new data frame object (`surveys_last`) from that last row
###
### 3. Use `nrow()` to extract the row that is in the middle of the
###    data frame. Store the content of this row in an object named
###    `surveys_middle`.
###
### 4. Combine `nrow()` with the `-` notation above to reproduce the behavior of
###    `head(surveys)`, keeping just the first through 6th rows of the surveys
###    dataset.

```


Factors

```{r, echo=FALSE, purl=TRUE}
### Factors
```

When we did `str(surveys)` we saw that several of the columns consist of
integers. The columns `genus`, `species`, `sex`, `plot_type`, ... however, are
of the class `character`.
Arguably, these columns contain categorical data, that is, they can only take on
a limited number of values.
```

```
R has a special class for working with categorical data, called `factor`.
Factors are very useful and actually contribute to making R particularly well
suited to working with data. So we are going to spend a little time introducing
them.

Once created, factors can only contain a pre-defined set of values, known as
*levels*.
Factors are stored as integers associated with labels and they can be ordered or unordered. While factor

When importing a data frame with `read_csv()`, the columns that contain text are not automatically coerc
loaded the data we can do the conversion using the `factor()` function:

```{r, purl=FALSE}
surveys$sex <- factor(surveys$sex)
```

We can see that the conversion has worked by using the `summary()`
function again. This produces a table with the counts for each factor level:

```{r, purl=FALSE}
summary(surveys$sex)
```

By default, R always sorts levels in alphabetical order. For
instance, if you have a factor with 2 levels:

```{r, purl=TRUE}
sex <- factor(c("male", "female", "female", "male"))
```

R will assign `1` to the level `"female"` and `2` to the level `"male"` (because
`f` comes before `m`, even though the first element in this vector is
`"male"`). You can see this by using the function `levels()` and you can find the
number of levels using `nlevels()`:

```{r, purl=FALSE}
levels(sex)
nlevels(sex)
```

Sometimes, the order of the factors does not matter, other times you might want
to specify the order because it is meaningful (e.g., "low", "medium", "high"),
it improves your visualization, or it is required by a particular type of
analysis. Here, one way to reorder our levels in the `sex` vector would be:

```{r, results=TRUE, purl=FALSE}
sex # current order
sex <- factor(sex, levels = c("male", "female"))
sex # after re-ordering
```

In R's memory, these factors are represented by integers (1, 2, 3), but are more
informative than integers because factors are self describing: `"female"`,
```

```
‘"male"‘ is more descriptive than ‘1‘, ‘2‘. Which one is "male"?  You wouldn’t
be able to tell just from the integer data. Factors, on the other hand, have
this information built in. It is particularly helpful when there are many levels
(like the species names in our example dataset).


> ### Challenge
>
> 1. Change the columns ‘taxa‘ and ‘genus‘ in the ‘surveys‘ data frame into a
>    factor.
>
> 2. Using the functions you learned before, can you find out...
>
>      * How many rabbits were observed?
>      * How many different genera are in the ‘genus‘ column?
>
> ‘‘‘{r, answer=TRUE, purl=FALSE}
> surveys$taxa <- factor(surveys$taxa)
> surveys$genus <- factor(surveys$genus)
> summary(surveys)
> nlevels(surveys$genus)
>
> ## * how many genera: There are 26 unique genera in the ‘genus‘ column.
> ## * how many rabbts: There are 75 rabbits in the ‘taxa‘ column.
> ‘‘‘

‘‘‘{r, echo=FALSE, purl=TRUE}
### Challenges:
###
### 1. Change the columns ‘taxa‘ and ‘genus‘ in the ‘surveys‘ data frame into a
###    factor.
###
### 2. Using the functions you learned before, can you find out...
###
###      * How many rabbits were observed?
###      * How many different genera are in the ‘genus‘ column?

‘‘‘


### Converting factors

If you need to convert a factor to a character vector, you use
‘as.character(x)‘.

‘‘‘{r, purl=FALSE}
as.character(sex)
‘‘‘

In some cases, you may have to convert factors where the levels appear as
numbers (such as concentration levels or years) to a numeric vector. For
instance, in one part of your analysis the years might need to be encoded as
factors (e.g., comparing average weights across years) but in another part of
```

```
your analysis they may need to be stored as numeric values (e.g., doing math
operations on the years). This conversion from factor to numeric is a little
trickier. The `as.numeric()` function returns the index values of the factor,
not its levels, so it will result in an entirely new (and unwanted in this case)
set of numbers. One method to avoid this is to convert factors to characters,
and then to numbers.

Another method is to use the `levels()` function. Compare:

```{r, purl=TRUE}
year_fct <- factor(c(1990, 1983, 1977, 1998, 1990))
as.numeric(year_fct)               # Wrong! And there is no warning...
as.numeric(as.character(year_fct)) # Works...
as.numeric(levels(year_fct))[year_fct]    # The recommended way.
```

Notice that in the `levels()` approach, three important steps occur:

* We obtain all the factor levels using `levels(year_fct)`
* We convert these levels to numeric values using `as.numeric(levels(year_fct))`
* We then access these numeric values using the underlying integers of the
  vector `year_fct` inside the square brackets

Renaming factors

When your data is stored as a factor, you can use the `plot()` function to get a
quick glance at the number of observations represented by each factor
level. Let's look at the number of males and females captured over the course of
the experiment:

```{r, purl=TRUE}
## bar plot of the number of females and males captured during the experiment:
plot(surveys$sex)
```

However, as we saw when we used `summary(surveys$sex)`, there are about 1700
individuals for which the sex information hasn't been recorded. To show them in
the plot, we can turn the missing values into a factor level with the
`addNA()` function. We will also have to give the new factor level a label.
We are going to work with a copy of the `sex` column, so we're not modifying the
working copy of the data frame:

```{r, results=TRUE, purl=FALSE}
sex <- surveys$sex
levels(sex)
sex <- addNA(sex)
levels(sex)
head(sex)
levels(sex)[3] <- "undetermined"
levels(sex)
head(sex)
```
```

```
Now we can plot the data again, using `plot(sex)`.

```{r echo=FALSE, purl=FALSE, results=TRUE}
plot(sex)
```

> ### Challenge
>
> * Rename "F" and "M" to "female" and "male" respectively.
> * Now that we have renamed the factor level to "undetermined", can you recreate the barplot such that
>
> ```{r, answer=TRUE, purl=FALSE}
> levels(sex)[1:2] <- c("female", "male")
> sex <- factor(sex, levels = c("undetermined", "female", "male"))
> plot(sex)
> ```


```{r wrong-order, results='show', echo=FALSE, purl=TRUE}
## Challenges
##
## * Rename "F" and "M" to "female" and "male" respectively.
## * Now that we have renamed the factor level to "undetermined", can you recreate the
##   barplot such that "undetermined" is first (before "female")
```


> ### Challenge
>
> 1. We have seen how data frames are created when using `read_csv()`, but
>   they can also be created by hand with the `data.frame()` function. There are
>   a few mistakes in this hand-crafted `data.frame`. Can you spot and fix them?
>   Don't hesitate to experiment!
>
>     ```{r, eval=FALSE, purl=FALSE}
>     animal_data <- data.frame(
>               animal = c(dog, cat, sea cucumber, sea urchin),
>               feel = c("furry", "squishy", "spiny"),
>               weight = c(45, 8 1.1, 0.8)
>               )
>     ```
>
>     ```{r, eval=FALSE, purl=TRUE, echo=FALSE}
>     ## Challenge:
>     ##  There are a few mistakes in this hand-crafted `data.frame`,
>     ##  can you spot and fix them? Don't hesitate to experiment!
>     animal_data <- data.frame(
>           animal = c(dog, cat, sea cucumber, sea urchin),
>           feel = c("furry", "squishy", "spiny"),
>           weight = c(45, 8 1.1, 0.8)
>           )
>     ```
>
```

```
> 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 you need to change to ensure that each column had the accurate data type?
>
>     ```{r, eval=FALSE, purl=FALSE}
>     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, eval=FALSE, purl=TRUE, echo=FALSE}
>    ## Challenge:
>    ##   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 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))
>    ```
>
>    ```{text_answer, echo=FALSE, purl=FALSE}
>    * missing quotations around the names of the animals
>    * missing one entry in the `feel` column (probably for one of the furry animals)
>    * missing one comma in the `weight` column
>    * `country`, `climate`, `temperature`, and `northern_hemisphere` are
>       characters; `has_kangaroo` is numeric
>    * using `factor()` one could replace character columns with factors columns
>    * removing the quotes in `temperature` and `northern_hemisphere` and replacing 1
>      by TRUE in the `has_kangaroo` column would give what was probably
>      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 (for instance, a letter in a column that should only contain numbers).

Learn more in this [RStudio tutorial](https://support.rstudio.com/hc/en-us/articles/218611977-Importing-

Formatting dates

A common issue that new (and experienced!) R users have is
```

```
converting date and time information into a variable that is suitable for
analyses. One way to store date information is to store each component of the
date in a separate column. Using `str()`, we can confirm that our data frame
does indeed have a separate column for day, month, and year, and that each of
these columns contains integer values.

```{r, eval=FALSE, purl=FALSE}
str(surveys)
```

We are going to use the `ymd()` function from the package **`lubridate`** (which belongs to the **`tidyv

Start by loading the required package:

```{r load-package, message=FALSE, purl=FALSE}
library(lubridate)
```

The **`lubridate`** package has many useful functions for working with dates.
These can help you extract dates from different string representations,
convert between timezones, calculate time differences and more. You can find
an overview of them in the [lubridate cheat sheet](https://raw.githubusercontent.com/rstudio/cheatsheets

Here we will use the function `ymd()`, which takes a vector representing year,
month, and day, and converts it to a `Date` vector.
`Date` is a class of data recognized by R as being a date and can
be manipulated as such. The argument that the function requires is flexible,
but, as a best practice, is a character vector formatted as "YYYY-MM-DD".

Let's create a date object and inspect the structure:

```{r, purl=FALSE}
my_date <- ymd("2015-01-01")
str(my_date)
```

Now let's paste the year, month, and day separately - we get the same result:

```{r, purl=FALSE}
# sep indicates the character to use to separate each component
my_date <- ymd(paste("2015", "1", "1", sep = "-"))
str(my_date)
```

Now we apply this function to the surveys dataset. Create a character vector from the `year`, `month`, a
`surveys` using `paste()`:

```{r, purl=FALSE}
paste(surveys$year, surveys$month, surveys$day, sep = "-")
```

This character vector can be used as the argument for `ymd()`:

```{r, purl=FALSE}
```
```

```
ymd(paste(surveys$year, surveys$month, surveys$day, sep = "-"))
```

There is a warning telling us that some dates could not be parsed (understood)
by the `ymd()` function. For these dates, the function has returned `NA`, which
means they are treated as missing values.
We will deal with this problem later, but first we add the resulting `Date`
vector to the `surveys` data frame as a new column called `date`:

```{r, purl=FALSE}
surveys$date <- ymd(paste(surveys$year, surveys$month, surveys$day, sep = "-"))
str(surveys) # notice the new column, with 'date' as the class
```

Let's make sure everything worked correctly. One way to inspect the new column is to use `summary()`:

```{r, results=TRUE, purl=FALSE}
summary(surveys$date)
```

Let's investigate why some dates could not be parsed.

We can use the functions we saw previously to deal with missing data to identify
the rows in our data frame that are failing. If we combine them with what we learned about subsetting da

```{r, results=TRUE, purl=FALSE}
missing_dates <- surveys[is.na(surveys$date), c("year", "month", "day")]

head(missing_dates)
```

Why did these dates fail to parse? If you had to use these data for your
analyses, how would you deal with this situation?

The answer is because the dates provided as input for the `ymd()` function do not actually exist. If we

There are several ways you could deal with situation:
* If you have access to the raw data (e.g., field sheets) or supporting information (e.g., field trip re
* If you are able to contact the person responsible for collecting the data, you could refer to them and
* You could also check the rest of the dataset for clues about the correct value for the erroneous dates
* If your project has guidelines on how to correct this sort of errors, refer to them and apply any reco
* If it is not possible to ascertain the correct value for these observations, you may want to leave the

Regardless of the option you choose, it is important that you document the error and the corrections (if

```{r, child="_page_built_on.Rmd"}
```

```
## Error:  attempt to use zero-length variable name
```

The R session information (including the OS info, R version and all packages used):

```
sessionInfo()

## R version 4.2.1 (2022-06-23 ucrt)
```

```
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19045)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_New Zealand.utf8  LC_CTYPE=English_New Zealand.utf8
## [3] LC_MONETARY=English_New Zealand.utf8 LC_NUMERIC=C
## [5] LC_TIME=English_New Zealand.utf8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] ggrepel_0.9.2      ggthemes_4.2.4     sf_1.0-9           leaflet_2.1.1
##  [5] fpp_0.5            tseries_0.10-53    lmtest_0.9-40      zoo_1.8-11
##  [9] expsmooth_2.3      fma_2.4            forecast_8.20      seasonal_1.9.0
## [13] lubridate_1.9.1    timetk_2.8.2       yardstick_1.1.0    workflowsets_1.0.0
## [17] workflows_1.1.2    tune_1.0.1         rsample_1.1.1      recipes_1.0.4
## [21] parsnip_1.0.3      modeldata_1.1.0    infer_1.0.4        dials_1.1.0
## [25] scales_1.2.1       broom_1.0.3        tidymodels_1.0.0   modeltime_1.2.4
## [29] flextable_0.8.5    XKCDdata_0.1.0     forcats_1.0.0      stringr_1.5.0
## [33] dplyr_1.1.0        purrr_1.0.1        readr_2.1.3        tidyr_1.3.0
## [37] tibble_3.1.8       ggplot2_3.4.0      tidyverse_1.3.2    knitr_1.42
##
## loaded via a namespace (and not attached):
##   [1] utf8_1.2.3           tidyselect_1.2.0     htmlwidgets_1.6.1    grid_4.2.1
##   [5] munsell_0.5.0        units_0.8-1          codetools_0.2-18     xgboost_1.7.3.1
##   [9] future_1.31.0        withr_2.5.0          colorspace_2.1-0     highr_0.10
##  [13] uuid_1.1-0           rstudioapi_0.14      stats4_4.2.1         wk_0.7.1
##  [17] officer_0.5.2        TTR_0.24.3           listenv_0.9.0        labeling_0.4.2
##  [21] rstan_2.21.8         DiceDesign_1.9       farver_2.1.1         parallelly_1.34.0
##  [25] vctrs_0.5.2          generics_0.1.3       ipred_0.9-13         xfun_0.37
##  [29] timechange_0.2.0     R6_2.5.1             lhs_1.1.6            cachem_1.0.6
##  [33] assertthat_0.2.1     promises_1.2.0.1     nnet_7.3-17          googlesheets4_1.0.1
##  [37] gtable_0.3.1         globals_0.16.2       processx_3.8.0       timeDate_4022.108
##  [41] rlang_1.0.6          systemfonts_1.0.4    splines_4.2.1        lazyeval_0.2.2
##  [45] gargle_1.3.0         inline_0.3.19        s2_1.1.2             yaml_2.3.7
##  [49] modelr_0.1.10        crosstalk_1.2.0      backports_1.4.1      httpuv_1.6.8
##  [53] quantmod_0.4.20      tools_4.2.1          lava_1.7.1           ellipsis_0.3.2
##  [57] proxy_0.4-27         Rcpp_1.0.10          base64enc_0.1-3      classInt_0.4-8
##  [61] ps_1.7.2             prettyunits_1.1.1    rpart_4.1.16         openssl_2.0.5
##  [65] fracdiff_1.5-2       haven_2.5.1          fs_1.6.0             tinytex_0.44
##  [69] furrr_0.3.1          crul_1.3             magrittr_2.0.3       data.table_1.14.6
##  [73] reprex_2.0.2         GPfit_1.0-8          googledrive_2.0.0    x13binary_1.1.57-3
##  [77] matrixStats_0.63.0   hms_1.1.2            mime_0.12            evaluate_0.20
##  [81] xtable_1.8-4         readxl_1.4.1         gridExtra_2.3        compiler_4.2.1
##  [85] KernSmooth_2.23-20   crayon_1.5.2         StanHeaders_2.21.0-7 htmltools_0.5.4
##  [89] later_1.3.0          tzdb_0.3.0           RcppParallel_5.1.6   DBI_1.1.3
##  [93] dbplyr_2.3.0         MASS_7.3-57          Matrix_1.5-3         cli_3.6.0
##  [97] quadprog_1.5-8       parallel_4.2.1       gower_1.0.1          pkgconfig_2.0.3
## [101] plotly_4.10.1        xml2_1.3.3           foreach_1.5.2        hardhat_1.2.0
## [105] prodlim_2019.11.13   rvest_1.0.3          snakecase_0.11.0     callr_3.7.3
```

```
## [109] digest_0.6.31        janitor_2.2.0        httpcode_0.3.0       rmarkdown_2.20
## [113] cellranger_1.1.0     gdtools_0.3.0        curl_5.0.0           shiny_1.7.4
## [117] urca_1.3-3           lifecycle_1.0.3      nlme_3.1-157         jsonlite_1.8.4
## [121] viridisLite_0.4.1    askpass_1.1          fansi_1.0.4          pillar_1.8.1
## [125] lattice_0.20-45      loo_2.5.1            fastmap_1.1.0        httr_1.4.4
## [129] pkgbuild_1.4.0       survival_3.3-1       glue_1.6.2           xts_0.12.2
## [133] zip_2.2.2            iterators_1.0.14     class_7.3-20         stringi_1.7.12
## [137] prophet_1.0          gfonts_0.2.0         memoise_2.0.1        e1071_1.7-13
## [141] future.apply_1.10.0

Sys.time()

## [1] "2023-02-03 13:23:41 NZDT"
```