Working with Data Frames

YOUR NAME HERE

2026-01-13

In this lesson we will learn how to summarize data in a data frame, and to do basic data management tasks such as making new variables, recoding data and dealing with missing data.

```
library(_______)                    # this came installed with tidyverse
library(_______)                   # for fancy summary tables
ncbirths <- openintro::ncbirths
```

```
Error in parse(text = input): <text>:1:10: unexpected input
1: library(__
             ^
```

Summarizing data

Two common methods used to summarize data are frequnecy tables for categorical variables (e.g. nominal, ordinal), and summary statistics for numeric (continuous or discrete) variables.

Frequency Tables

Frequency tables are used only any type of categorical data (Nominal, ordinal or binary), and the table results show you how many records in the data set have that particular level.

You can create a basic frequency table by using the `table()` function.

```
_______(ncbirths$lowbirthweight)
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

Relative frequencies (proportions or percentages) are calculated by putting the results of the `table` function inside the `prop_table` function.

```
_______(
  table(ncbirths$lowbirthweight)
)
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

The variable `ncbirths$lowbirthweight` has _____ records with a value of `low`, and _____ records with the value of `not low`.

Summary Statistics

Numerical variables can be summarized using quantities called *summary statistics* which include the `min`, `max`, `mean` and median. The function `summary()` prints out the five number summary, and includes the mean. This function also displays the number of missing values for that variable.

```
_______(ncbirths$visits)
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

There are also individual functions available

```
_______(ncbirths$mage)
_______(ncbirths$mage)
_______(ncbirths$mage)
max(ncbirths$mage)
min(ncbirths$mage)
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

Fancy summary tables

The `gtsummary` package provides a single function `tbl_summary` to create a really nicely formatted summary table for both quantitative and categorical data types.

The first argument is the data set, then you `include` the vector of variable that you want to display in the table. By default the sample size n and the relative percent are presented for categorical data, and the median, with first and third quartiles shown for quantitiatve data.

```
tbl_summary(ncbirths,
            _______ = c(_______, _______)
            )
```

```
Error in parse(text = input): <text>:2:14: unexpected input
1: tbl_summary(ncbirths,
2:              __
                ^
```

The `statistic` argument can be used to change what values are displayed. Here we are specifying we want the mean `{mean}` and standard deviation `{sd}` for all variables that R sees as continuous (numeric), and both the frequency `{n}`, the total number of non-missing values `{N}` and the percent `{p}` for each level of a categorical variable.

```
_______(ncbirths,
            include = c(visits, lowbirthweight),
            _______ = list(
                _______() ~ "{mean} ({sd})",
                _______() ~ "{n} / {N} ({p}%)"
              )
            )
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

Missing Data

Sometimes the value for a variable is missing. Think of it as a blank cell in an spreadsheet. Missing data can be a result of many things: skip patterns in a survey (i.e. non-smokers don't get asked how many packs per week they smoke), errors in data reads from a machine, researchers skipped a day of data collection for one plant on accident etc.

R puts a `NA` as a placeholder when the value for that piece of data is missing. We can see 4 out of the first 6 values for the variable `fage` (fathers age) in the `ncbirths` data set are missing.

```
_______(ncbirths$fage)
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

Problem 1: `R` can't do arithmetic on missing data.

So `5 + NA = NA`, and if you were to try to calculate the `mean()` of a variable, you'd also get `NA`.

```
_______(ncbirths$fage)
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

Fix this error

Add the argument `na.rm=TRUE` to the `mean()` function inside the right hand parenthesis to calculate the mean after excluding missing values.
Run this code chunk interactively to ensure that it works before continuing.

```
mean(ncbirths$fage, _______)
```

```
Error in parse(text = input): <text>:1:22: unexpected input
1: mean(ncbirths$fage, __
                       ^
```

Problem 2: Some plots will show `NA` as it's own category

Sometimes this is fine, other times this is undesirable. We'll see later how we can adjust this plot to remove that column of NA.

```
ggplot(ncbirths, aes(premie)) + geom_bar()
```

```
Error in ggplot(ncbirths, aes(premie)): could not find function "ggplot"
```

Missing values can cause some problems during analysis or undesirable features in a plot so let's see how to detect missing values and how to work around them.

Identifying missing values

To find out how many values in a particular variable are missing we can use several different approaches.

Look at the raw data

We can look at the raw data using `head()` or opening the data set in the spreadsheet view and skim with our eyes for `NA` values. This may not be helpful if there is no missing values in the first 6 rows, or if there is a large number of variables to look through.

```
_______(ncbirths)
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

Look at data summaries

Functions such as `table()` have a `useNA="always"` option to show how many records have missing values, and `summary()` will always show a column for NA.

```
table(ncbirths$habit, _______)
_______(ncbirths$fage)
```

```
Error in parse(text = input): <text>:1:24: unexpected input
1: table(ncbirths$habit, __
                           ^
```

Use a logical statement

The function `is.na()` returns TRUE or FALSE for each element in the provided vector for whether or not that element is missing.

```
x <- c("green", NA, 3)
_______(x)
```

```
Error in parse(text = input): <text>:2:2: unexpected input
1: x <- c("green", NA, 3)
2: __
     ^
```

In this example, the vector `x` is created with three elements, the second one is missing. Calling the function `is.na()` on the vector `x`, results in three values, where only the second one is TRUE – meaning the second element is missing.

This can be extended to do things such as using the `sum()` function to count the number of missing values in a variable. Here we are *nesting* the functions `is.na()` is written entirely inside the `sum()` function.

```
_______(_______(ncbirths$fage))
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

There are 171 records in this data set where the age for the father is not present.

> **Negating `is.na()` to find the non-missing values**
>
> Sometimes you want to operate only only the non-missing values. Recall from earlier we can use the `!` to negate a boolean argument.

```
_______(x)
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

The first and third values are TRUE - so they are **not** missing.

> **Hide the NA bar**
>
> We can use this tactic to fix that barchart from above. Wrap `!is.na()` around the `premie` variable in the `ggplot` code to create a barchart that does not have a bar for missing values.
>
> ```
> ggplot(ncbirths, aes(_______(premie))) + geom_bar()
> ```
>
> ```
> Error in parse(text = input): <text>:1:23: unexpected input
> 1: ggplot(ncbirths, aes(__
>                           ^
> ```

Data management

Sometimes we have a need to create or modify variables in a data frame. You will learn several ways to do this throughout this course, but we will start by using *base R* functions and methods. These are methods that use functions that come with R, not from additional packages.

Overwrite existing values

Choose all observations (rows) of a `data` set, where a `variable` is equal to some `value`, then set assign `<-` a `new_value` to those rows.

```
data[data$variable==value] <- new_value  # example code to show syntax.
```

> Example: Too low birthweight
>
> Let's look at the numerical distribution of birthweight (in pounds) of the baby.
>
> ```
> summary(_______)
> ```
>
> ```
> Error in parse(text = input): <text>:1:10: unexpected input
> 1: summary(__
>              ^
> ```

The value of 1 lb seems very low. The researchers you are working with decide that is a mistake and should be excluded from the data. We would then set all records where `weight=1` to missing.

```
ncbirths$_______[ncbirths$_______] <- _______
```

```
Error in parse(text = input): <text>:1:10: unexpected input
1: ncbirths$_
             ^
```

Code explainer:

- The specific variable `ncbirths$weight` is on the left side outside the `[]`. So just the variable `weight` is being changed.
- Recall that bracket notation `[]` can be used to select rows where a certain logical statement is true. So `[ncbirths$weight==1]` will only show records where `weight` is equal to 1.

- Notice where the assignment arrow (`<-`) is at. This code assigns the value of `NA` (missing) to the variable `weight`, where `weight==1`.

```
_______(_______, na.rm=TRUE)
```

```
Error in parse(text = input): <text>:1:2: unexpected input
1: __
     ^
```

The minimum weight is now 1.19.

> **Your Turn**
>
> But what about other weights that aren't quite as low as 1, but still unusually low?
>
> - Write the code to set all birth weights less than 4 lbs (`<4`) to missing (NA).
> - Then recalculate the mean to confirm your recode worked.

Creating new variables

> **! New variables should be added to the data frame**
>
> This can be done in base R using `$` sign notation.

The new variable you want to create goes on the left side of the assignment operator `<-`, and how you want to create that new variable goes on the right side.

```
data$new_variable <- creation_statement  # example code not run
```

> **Example: Row-wise difference between two existing variables**
>
> As a pregnancy progresses, both the mother and the baby gain weight. The variable `gained` is the total amount of weight the mother gained in her pregnancy. The variable `weight` is how much the baby weighed at birth.

The following code creates a new variable `wtgain_mom` the weight gained by the mother, that is not due to the baby by subtracting `weight` from `gained`. Note all variables are prefaced with `$`, denoting that they exist inside the `ncbirths` data set.

```
ncbirths$_______ <- ncbirths$_______ - ncbirths$_______
```

```
Error in parse(text = input): <text>:1:10: unexpected input
1: ncbirths$_
             ^
```

To confirm this variable was created correctly, we look at the data contained in three variables in question.

```
head(ncbirths[,c('_______', '_______', '_______')])
```

```
Error: object 'ncbirths' not found
```

> **!** Trust but Verify
>
> It's always important to visually confirm that the code you wrote actually had the intended effect.

Dichtomizing data

The `ifelse()` is hands down the easiest way to create a binary variable (dichotomizing, only 2 levels)

Let's add a variable to identify if a mother in the North Carolina births data set was underage at the time of birth. Specifically Make a new variable `underage` on the `ncbirths` data set. If `mage` is under 18, then the value of this new variable is `underage`, else it is labeled as `adult`.

```
ncbirths$_______ <- _______(ncbirths$_______, "_______", "_______")
```

```
Error in parse(text = input): <text>:1:10: unexpected input
1: ncbirths$_
             ^
```

Code explainer:

- The function is `ifelse()` - one word.
- The arguments are: `ifelse(logical, value if TRUE, value if FALSE)`
  - The `logical` argument is a statement that resolves as a `boolean` variable, as either TRUE or FALSE.

- The second argument is what you want the resulting variable to contain if the logical argument is `TRUE`.
- The last argument is what you want the resulting variable to contain if the logical argument is `FALSE`.

Trust but Verify

First let's look at the frequency table of `underage` and see if records exist with the new categories, and if there are any missing values.

```
table(_______, useNA="always")
```

```
Error in parse(text = input): <text>:1:8: unexpected input
1: table(__
           ^
```

Next let's check it against the value of `mage` itself. Let's look at all rows where mothers age is either 17 or 18 `mage %in% c(17,18)`, and only the columns of interest.

```
ncbirths[ncbirths$mage %in% c(17,18),c('mage', 'underage')]
```

```
Error: object 'ncbirths' not found
```

Notice I snuck a new operator in on you - `%in%`. This is a way you can provide a list of values (a.k.a a vector) and say "if the value of the variable I want is %in% any of these options in this vector…" do the thing.

Chaining commands

Two common styles:

- `|>` This is the "native" pipe that's built into R.
- `%>%` This pipe is loaded with the `tidyverse` package.

They both function the same, but you'll see both being used so it's good to know that they both exist. That way if you are not using other functions from the `tidyverse`, you can still enjoy the chaining functionality.

What is "Chaining"?

The pipe lets you string set of functions together, like links on a chain, to be completed in the order specified. This works with the majority of functions, specifically when the result of the function is a data frame, a vector, or sometimes the results of a model.

> "and then…."

This is what I read to myself when using the pipe. "Do this `|>` the next thing `|>` do this third thing `|>` this last"

Example: Frequency tables & summary statistics using the pipe

First stating the variable, then pipe in the summary function.

```
ncbirths$mature |> _______() # instead of table(ncbirths$mature)
ncbirths$mage |> _______() #instead of mean(ncbirths$mage)
```

```
Error in "_": The pipe operator requires a function call as RHS (<input>:1:20)
```

These can be read as

1. Get the `mature` variable from the `ncbirths` data set
2. *and then* create a frequency table on that variable

and

1. Get the `mage` variable from the `ncbirths` data set
2. *and then* calculate the mean of that variable

These may be trivial examples now but the usefulness of this approach will be apparent before the class is finished.

! Behind the scenes

What actually is happening, is that the result from the code on the left of the `|>` gets passed into the first argument of the commands on the right hand side. Two things to keep in mind:

1. Do not include the variable on both sides.

```
ncbirths$mature |>  # do this
  table()

ncbirths$mature |>
  table(ncbirths$mature) # not this
```

2. the pipe itself must be at the end of a "sentence".

```
ncbirths$mature |>   # do this
  table()

ncbirths$mature      # not this
|> table()
```