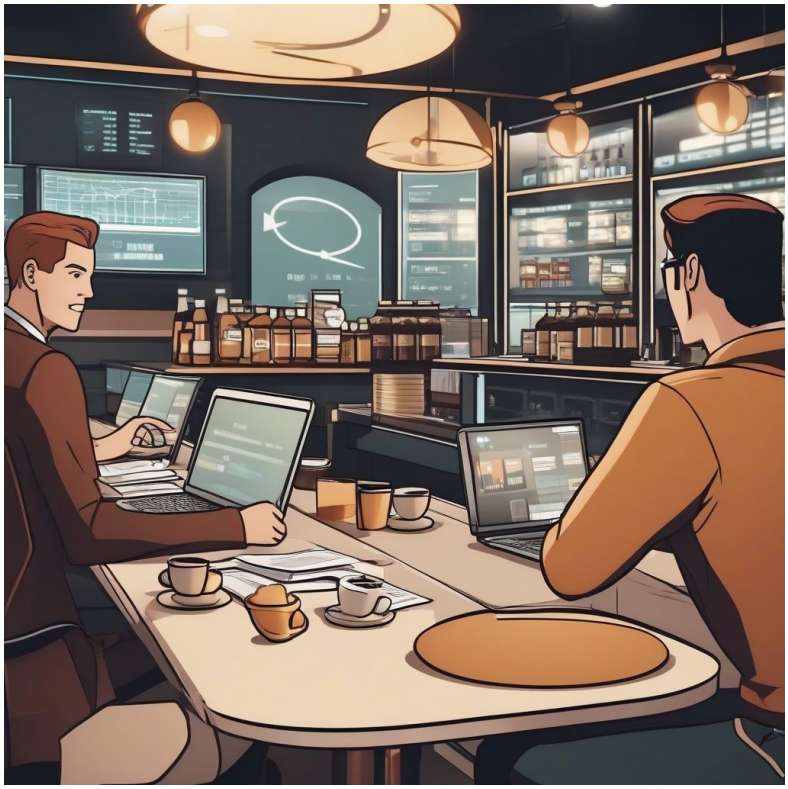
The data-wrangling and omics course for R novices

Casper-Emil Tingskov Pedersen and Jonathan Thorsen

2023-11-03

Welcome to this course



This course will introduce you to data-wrangling and analysis of omics data. We start out by building an intuition for working with data in R/Rstudio and then plot our results.

The best way to learn is to follow along with your own laptop, but all are welcome. The idea with this course is also that you can do your own selv-paced learning by going back to themes that are harder.

We'll spend half the time with the lectures on the basic of R and data wrangling and half the time for you to try out wrangling yourself. You can also try some

of the things you learn from the exercises on your own data.

Before you begin, be sure you are all set up: see the prerequisites in #overview.

Breakdown for the course:

Course schedule:

Time	Day1	Day2
9-10:30	Data-Wrangling, an introduction	Modelling introduction and omics
10:35-12	Get your hands dirty with data-wrangling	Your own catwalk with omics
13-14.30	GGplot2, an introduction	Machine learning introduction
14.35-16	Plotting visually please plots	Omics and machine learning

---

Course overview

Yihaa you are commited to learning relevant coding using R. The world is now ready to become your oister.

What can you expect

In this course you will learn how to code in R and it will be fun. You will learn efficient code and workflows that can be used in your own research and for various kinds of data, including all types of omics data. This is really powerful and will lead you to become a better scientist.

General learning outcomes

-how to code in R.

-how to THINK about data.

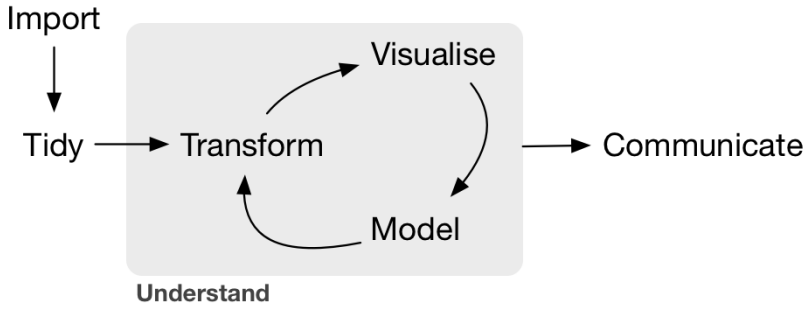
-how to think about data separately from your research questions.

-how and why to tidy data and analyze tidy data.

-how to increase efficiency in your research.

Our workflow plus the Tidy data workflow

We will turning our data tidy and how to use a tidyverse suite of tools to work with tidy data.
It has been developed by Hadley Wickham and his team, please visit his website if you are interested in learning more.



We will be focusing on a combination of tools:

Basic tools: -**str/glimpse**: Look at data

Tidytools:

-**Tidy**: `tidyr` to organize rows of data into unique values.

-**Transform**: `dplyr` to manipulate/wrangle data based on subsetting by rows or columns, sorting and joining.

-**Visualize**: `ggplot2` static plots, using grammar of graphics principles

This is essential - Instead of building your analyses around the format your data are in, take deliberate steps to make your data tidy. When your data are tidy, you can use a growing suite of powerful analytical and visualization tools instead of inventing home-grown ways to accommodate your data. This will save you time since you aren't reinventing the wheel, and will make your work more clear and understandable to your collaborators (most importantly, Future You).

Working with data that is not your own

One of the most important things you will learn is how to think about data separately from your own research context. Said in another way, you'll learn to distinguish your data questions from your research questions. Here, we are focusing on data questions, and we will use data that is not specific to your research.

We will be using several different data sets throughout this training, and will help you see the patterns and parallels to your own data, which will ultimately help you in your research.

Goals of this course

The goal of this course today is to equip you with the tools to take a (your) dataset and do the following:

1. Import data to R and understand the structure of your data
2. Look at it using summaries and descriptive statistics
3. Engineer features relevant for your research
4. Impute missing data
5. To do transformation and understand what that means
6. Plot the data using visually appealing and publish-ready figures

Prerequisites

A few things need to be checked before we start. First, we assume that you have either R or RStudio installed.

Also, if not already installed, please install the following packages: c("dplyr","tidyverse","broom"," ")

Credit

This material builds from (and is directly copied from) a lot of fantastic materials developed by others in the open data science community. In particular, it pulls from the following resources, which are highly recommended for further learning and as resources later on. Specific lessons will also cite more resources.

OHI data science training

Modern dive

Super quick intro to coding and data in R

Coding/programming in R

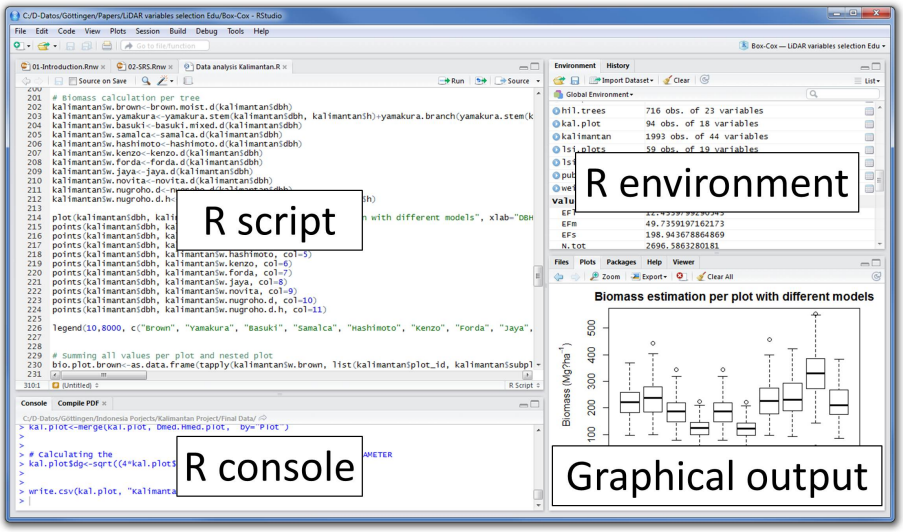
Now that you're set up with R and RStudio, you are probably asking yourself, "OK. Now how do I use R?". The first thing to note is that unlike other statistical software programs like Excel, SPSS, or Minitab that provide point-and-click interfaces, R is an interpreted language. This means you have to type in commands written in R code. In other words, you have to code/program in

R. Note that we'll use the terms "coding" and "programming" interchangeably throughout this course.

While it is not required to be a seasoned coder/computer programmer to use R, there is still a set of basic programming concepts that new R users need to understand. Consequently, while this course is not focused on learning programming, you will still learn just enough of these basic programming concepts needed to explore and analyze data effectively.

Basic terminology in R

We now introduce some basic programming concepts and terminology. Instead of asking you to memorize all these concepts and terminology right now, we'll guide you so that you'll "learn by doing." To help you learn, we will always use a different font to distinguish regular text from computer_code. Also we will go through a tiny bit of information and then ask you do a similar task on your computer.



On this plot you can see the overview of Rstudio. This contains the following elements:

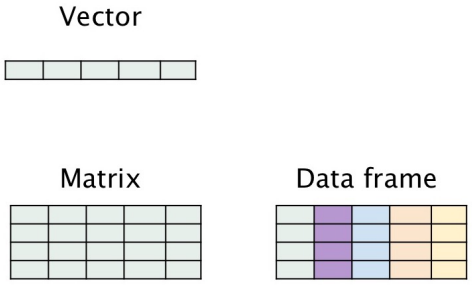
Console pane: where you enter in commands.

R script: Where you store your commands often in a text file.

R environment: What objects are registered/loaded in the environment (we will get back to that).

Graphical output: Where you can see figures that is produced by commands in the console.

Variables	Example
integer	100
numeric	0.05
character	“hello”
logical	TRUE
factor	“Green”



This figure shows some of the more important data types in R and how they relate to storing data in R. It seems simple when set up like this, but when you are coding and you need to find an error this figure is a good place to start.

Vectors can be created by using the `c()` function in R. The c stands for concatenate. for example the following command creates a vector of integers 1,2,3,4: `my_vector <- c(1,2,3,4)`

Matrices can be created by using the `matrix()` function in R. Matrices are great for doing calculations incredibly fast BUT they can only contain one type of data unlike dataframes. For example the following command creates a matrix with the same four integers 1,2,3,4: `my_matrix <- matrix(c(1,2,3,4))` you can also rearrange the matrix so that it has two columns instead of one: `my_matrix <- matrix(c(1,2,3,4),ncol=2)` you can look at your matrix by typing `my_matrix` in the console.

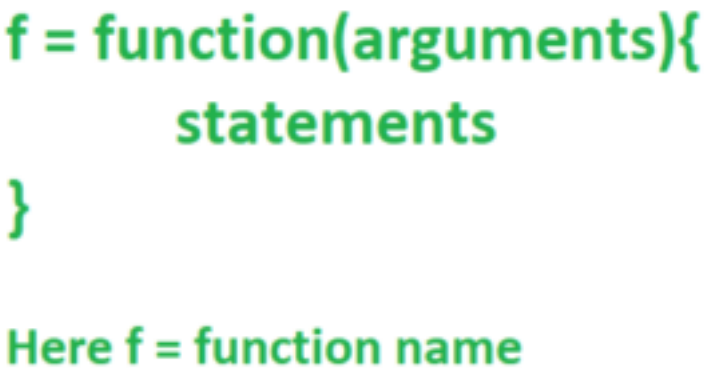
Data frames is a way to store data containing both strings, integers, factors together. They are rectangular spreadsheets. Typically you have a dataset where the rows correspond tot eh observations/samples and the columns correspond to the variables that describe the observations. And this is the typical format when we analyse data. We can create a data frame in R with mixed data like this: `my_df <- data.frame(int = c(1,2,3,4),names = c("Villy","Søren","Karl","Benny"))`.

	Operator	Definition
1	&	And
2	|	Or
3	!	Not
4	==	Equals (when checking a condition)
5	=	Equals (when defining a value)
6	!=	Not equals
7	>	Greater than
8	<	Less than
9	>=	Greater than or equal to
10	<=	Less than or equal to

Conditionals:

Often we subset our data using conditions, e.g. we only want samples of our data frame that has a value in column 3 that is larger than X. Do this kind of subsetting, we use *conditionals*.

For example, using the data frame from before, we can restrict rows to those that have a integer value larger than 2 like this: `my_df[my_df$int > 2,]` This will capture the last two rows of the data. Notice the comma, which is essential when indexing dataframes: everything before the comma is subsetting on rows while after the comma is for subsetting columns. We can also subset using equality using == (and NOT =, which is used for assignment). Here is an example using equality: `my_df[my_df$names == "Karl",]`.
The final example of something similar to but not identical is a binary operator %in% which is not listed in the figure above but still very useful. This weirdly looking %in% is describing a operator (which we will get back to and use later) which can search for overlaps e.g. in R: `my_df[my_df$names %in% c("Karl","Villy"),]` See here that we find the overlap between names in my_df and the provided vector and only showing the rows of the data frame where there is an overlap.



Functions, also called commands: Functions perform tasks in R. They take in inputs called arguments and return outputs. You can either manually specify a function's arguments or use the function's default values. For example, the function `seq()` in R generates a sequence of numbers. If you just run `seq()` it will return the value 1. That doesn't seem very useful! This is because the default arguments are set as `seq(from = 1, to = 1)`. Thus, if you don't pass in different values for from and to to change this behavior, R just assumes all you want is the number 1. You can change the argument values by updating the values after the = sign. If we try out `seq(from = 2, to = 5)` we get the result 2 3 4 5 that we might expect.

This list is by no means an exhaustive list of all the programming concepts and terminology needed to become a savvy R user; such a list would be so large it wouldn't be very useful, especially for novices. Rather, we feel this is a minimally viable list of programming concepts and terminology you need to know before getting started. We feel that you can learn the rest as you go. Remember that your mastery of all of these concepts and terminology will build as you practice more and more.

Your turn

Exercise 1.

1. Create a dataframe with the following information (you pick the dataframe name): two column dataframe. One with the names: `c("Villy","Søren","Karl","Benny","Siri")` and One with the following grades:`c(100,20,40,30,60)`.
2. Use the `mean()` function to calculate the mean for passing grades (>30). There are many ways to do this but a hint is that we only need to run one command. you can always see more information about the function by writing `?mean` in your console.

Error messages

One thing that intimidates new R and RStudio users is how it reports errors, warnings, and messages. R reports errors, warnings, and messages in a glaring red font, which makes it seem like it is scolding you. However, seeing red text in the console is not always bad.

R will show red text in the console pane in three different situations:

If the text starts with "Error", figure out what's causing it. Think of errors as a red traffic light: something is wrong! If the text starts with "Warning", figure out if it's something to worry about. For instance, if you get a warning about missing values in a scatterplot and you know there are missing values, you're fine. If that's surprising, look at your data and see what's missing. Think of warnings as a yellow traffic light: everything is working fine, but watch out/pay attention. Otherwise, the text is just a message. Read it, wave back at R, and thank it for talking to you. Think of messages as a green traffic light: everything is working fine and keep on going!

Coding tips and tricks

Learning to code/program is quite similar to learning a foreign language. It can be daunting and frustrating at first. Such frustrations are common and it is normal to feel discouraged as you learn. However, just as with learning a foreign language, if you put in the effort and are not afraid to make mistakes, anybody can learn and improve.

Here are a few useful tips to keep in mind as you learn to program:

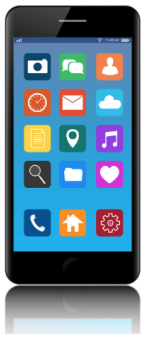
Remember that computers are not actually that smart: You may think your computer or smartphone is "smart," but really people spent a lot of time and energy designing them to appear "smart." In reality, you have to tell a computer everything it needs to do. Furthermore, the instructions you give your computer can't have any mistakes in them, nor can they be ambiguous in any way.

Take the "copy, paste, and tweak" approach: Especially when you learn your first programming language or you need to understand particularly complicated code, it is often much easier to take existing code that you know works and modify it to suit your ends. This is as opposed to trying to type out the code from scratch. We call this the "copy, paste, and tweak" approach. So early on, we suggest not trying to write code from memory, but rather take existing examples we have provided you, then copy, paste, and tweak them to suit your goals. After you start feeling more confident, you can slowly move away from this approach and write code from scratch. Think of the "copy, paste, and tweak" approach as training wheels for a child learning to ride a bike. After getting comfortable, they won't need them anymore.

The best way to learn to code is by doing: Rather than learning to code for its own sake, we find that learning to code goes much smoother when you have a goal in mind or when you are working on a particular project, like analyzing data that you are interested in and that is important to you.
Practice is key!

Install and load package

So we alluded to that the environment contains many objects and some of these objects come from packages. Packages in R is precalculated functions and useful tools to work with your data. Therefore, embrace packages and get familiar with some of them. Some of the more important ones we will use in this course are: `tidyverse` and `dplyr` for data massaging and wrangling and `ggplot 2` for visualizations.

R: A new phone	R Packages: Apps you can download
	

So R is like a new mobile phone: while it has a certain amount of features when you use it for the first time, it doesn't have everything. R packages are like the apps you can download onto your phone from Apple's App Store or Android's Google Play.

Like an app on your smartphone, you need to install it first and the open it.

if you try to load a package that is not installed you get an error for instance

`library(tidyverse)` and if not you know that tidyverse is installed

To install the tidyverse package do the following:

```
install.packages("tidyverse")
```

and to load it subsequently, write `library(tidyverse)`.

Now install `broom` , `tidyverse` and `dplyr` using this approach

Data wrangling and tidy data

"We're ready to scrub the data." #betterdata

Packages required

The following packages are required. If you need to go back and check how installing a package is done see the section 3.5(#installpackages).

library(dplyr).
library(ggplot2).
library(readr).
library(tidyr).

library(broom).
library(palmerpenguins).

This chapter will introduce you to data wrangling with an emphasis on the `dplyr` package along with other helpful functions. You will see how powerful wrangling can be and how it can be like a magic wand to transform data into your specific needs. The key functions include but are not limited to:

1. The overarching function which makes us able to combine functions and subsetting is **the pipe**. And what is a pipe? Pipes are a neat way to tie together several dataframes/functions into a chain of actions. It makes coding easier to understand and is therefore a better style of coding. It looks like this `%>%`. For instance taking the mean of the first two rows of the `my_df` dataframe can be done like so: `my_df[1:2,"Grades"] %>% mean`. Learning to use the pipe to your advantage is learning to code. Luckily it all comes naturally when working with the following functions:

2. `left_join()` merge two data frames together using a key column. This function (along with its cousin functions `right_join()` and `join()` etc.) is the bread and butter of working with many data types that needs to be merged and analysed. For example

```
library(dplyr)
my_data <- data.frame(
names = c("Villy", "Søren", "Karl", "Benny", "Siri"),
Grades = c(100, 20, 40, 30, 60),
gender = c("boy", "boy", "boy", "boy", "girl"))
other_data <- data.frame(gender = c("boy", "girl"), description = c("Male", "Female"))
left_joined_data <- left_join(my_data, other_data, by = "gender")

left_joined_data
```

```
##   names Grades gender description
## 1 Villy    100    boy        Male
## 2 Søren     20    boy        Male
## 3  Karl     40    boy        Male
## 4 Benny     30    boy        Male
## 5  Siri     60   girl      Female
```

1. `mutate()` construct/reconstruct a column using a newly defined column name. This function is useful to generate a new columns that modify existing ones often using a conditional to do so. Now using the same data as above:

```
  mutated_data <- my_data %>%
  mutate(Grades_scaled = Grades / max(Grades))
```

1. `glimpse()` take a sneak peak at a dataset. Useful for getting the dimensions of the data and the type of data contained in the columns. Now look at the newly constructed column

```
  glimpse(mutated_data)
```

```
## Rows: 5
## Columns: 4
## $ names         <chr> "Villy", "Søren", "Karl", "Benny", "Siri"
## $ Grades        <dbl> 100, 20, 40, 30, 60
## $ gender        <chr> "boy", "boy", "boy", "boy", "girl"
## $ Grades_scaled <dbl> 1.0, 0.2, 0.4, 0.3, 0.6
```

1. `count()` counts the n of each unique value in a column. Usefull to combine with mutate to see the counts of the newly constructed column.

```
  counted_data <- count(my_data, gender)
  counted_data
```

```
##   gender n
## 1    boy 4
## 2   girl 1
```

1. `arrange()` reorder the rows of a dataframe. For example, sort the rows of a dataset in descending order of the column height.

```
arranged_data <- arrange(my_data, desc(Grades))
arranged_data
```

```
##   names Grades gender
## 1 Villy    100    boy
## 2  Siri     60   girl
## 3  Karl     40    boy
## 4 Benny     30    boy
## 5 Søren     20    boy
```

1. `select()` select columns using indexing or strings. e.g. `my_df %>% select("names")`.

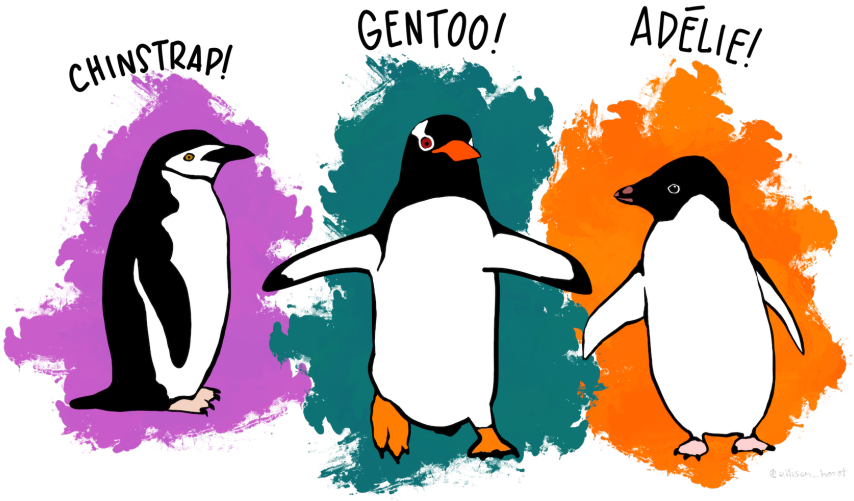
2. `filter()` the rows of a dataframe based on a filter, e.g. thresholds. It is similar to the example shown in the introduction, where we used old school filtering on my_df dataframe `my_df[my_df$names %in% c("Karl","Villy"),]`. Using the filter function and pipes you can do it like this: `my_df %>% filter(names %in% c("Karl","Villy"))`.

3. `summarise()` a dataframe using one or more of its columns/variables with a summary statistic. Such examples could be the median or the mean and an example look like this: `my_df %>% summarise(mean = mean(Grades))`. This function can be very powerful when combined with the `group_by` function.

4. `group_by()` is a dplyr way of grouping rows, it is a way to assign different rows to a group and do statistical reporting on every specific group in the data set. An example require us to add a third column to `my_df` like this: `my_df <- my_df %>% mutate(gender = c("boy","boy","boy","boy","girl"))`. Now using this new `my_df` we can `group_by` on the gender column and calculate the median on this group:

```
my_df %>%
  group_by(gender) %>%
  summarise(median(Grades))
```

```
## # A tibble: 2 x 2
##   gender `median(Grades)`
##   <chr>             <dbl>
## 1 boy                  35
## 2 girl                 60
```



Penguins!

Now we are going to use the functions described above on the penguin dataset. The data has been collected by Dr. Kirstin Gorman at the Palmer station, Antartica LTER. So real data collected for a purpose and published in a scientific journal.

```
library(palmerpenguins)
```

Your turn - Look at the dataset

Now look at the dataset using `glimpse()` in the console, now try to look at the dataset using `str()` again you get information about the dataset. Please use one minute and compare the two methods to look at structure of the data. Which one do you prefer and why?

Exercise 2.

1. You can input several grouping variables into `group_by(var1,var2)`. Now use this information to output the mean length of the flippers for species across islands, meaning for e.g. Adelie penguins you will have three values. Remember that `mean` can handle NA values in your dataset, you just have to tell it to do so.

More handy functions

Often, we are interested in a summary of information for many columns at once and thus we are not only looking at e.g. flipper_length across species and islands, but rather all measurements across species and islands. The reason to this more thorough data description is to gain better insight into the dataset. Better insights leads to better scientific questions.

1. `across()` is a function that makes it easy to transform/summarise multiple columns. It needs a *function* to apply to these columns, that could be `median` or `mean()`. Also, it is often combined with functions like `mutate()` and `summarise()` to generate new columns using transformation on existing columns. Furthermore, it is also often combined with functions that informs `across()` on which columns to look at. For instance, `across()` can be combined with `starts_with()` and `where()`. OK enough text, now to an example that makes sense. Again looking at the penguins dataset, lets try to combine these functions to get an intuition about how they work together:

```
penguins %>%
group_by(species,island) %>%
summarize(across(starts_with("bill"),~mean(., na.rm = TRUE)))
```

```
## `summarise()` has grouped output by 'species'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 5 x 4
## # Groups:   species [3]
##   species   island    bill_length_mm bill_depth_mm
##   <fct>     <fct>              <dbl>         <dbl>
## 1 Adelie    Biscoe              39.0          18.4
## 2 Adelie    Dream               38.5          18.3
## 3 Adelie    Torgersen           39.0          18.4
## 4 Chinstrap Dream               48.8          18.4
## 5 Gentoo    Biscoe              47.5          15.0
```

You can see that this gives a nice overview of the bill lengths across the species across the islands. One takeaway one this would be that the difference in bill length seems less influenced by islands and more influenced by species.

One way to deal with missing values

```
data <- penguins
sum(is.na(penguins))
```

```
## [1] 19
```

The penguins dataset is not perfect. BUT, we can wave our magic data wand to impute the missing values. What is imputation? it is best-guess construction, meaning we already know a lot about the penguins so we fill in the missing values with guesses. It is not cheating, it is done all the time in research.

```
# rename the dataset so we can modify the data frame without having to reload the data
data <- penguins
# how many missing in the flipper length?
sum(is.na(data$flipper_length_mm))
```

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

Now - waving the magic data wand - we impute

```
# Impute missing values with mean imputation for the 'flipper_length_mm' column
data_imputed <- data %>%
  mutate(flipper_length_mm = ifelse(is.na(flipper_length_mm),
                                    mean(flipper_length_mm, na.rm = TRUE),
                                    flipper_length_mm))
```

What is going on? Here we are being aided by the `ifelse` function which ask questions about the data, we pair this function with mutate and reconstruct the flipper_length_mm imputing missing values.

Typically though, we have a lot of variables that needs to be imputed and *even though we hate to admit it, we know that we are lazy, and laziness is the super fuel of the efficient programmer!* So below is a way to to the same, but instead of focusing on a specific column we use the `mutate_if` function to find all numeric columns in the dataset and do the same.

```
  # Impute missing values with mean imputation for numeric columns
  data_imputed <- data %>%
    mutate_if(is.numeric, ~ifelse(is.na(.), mean(., na.rm = TRUE), .))
```

Your turn

Exercise 3.

Sometimes the variables you need impute are not just continuous variables like height or flipper length in mm. So what do we do then? Well, we guess, but instead of using the information in the column you want to impute, you use supplementary information in the adjacent columns. How can we do that? Well consider the column in the penguins data set which contains the gender of the penguins. We would not gain anything from replacing NA values with the mean of that column. In the natural world, species are often sexually dimorphic, which are fancy words for differences in size/color between sexes. We want to leverage this information to guess at the gender. So we can guess based on body size because we know sexes often differ in size. But first, we need to check if that is true for penguins.

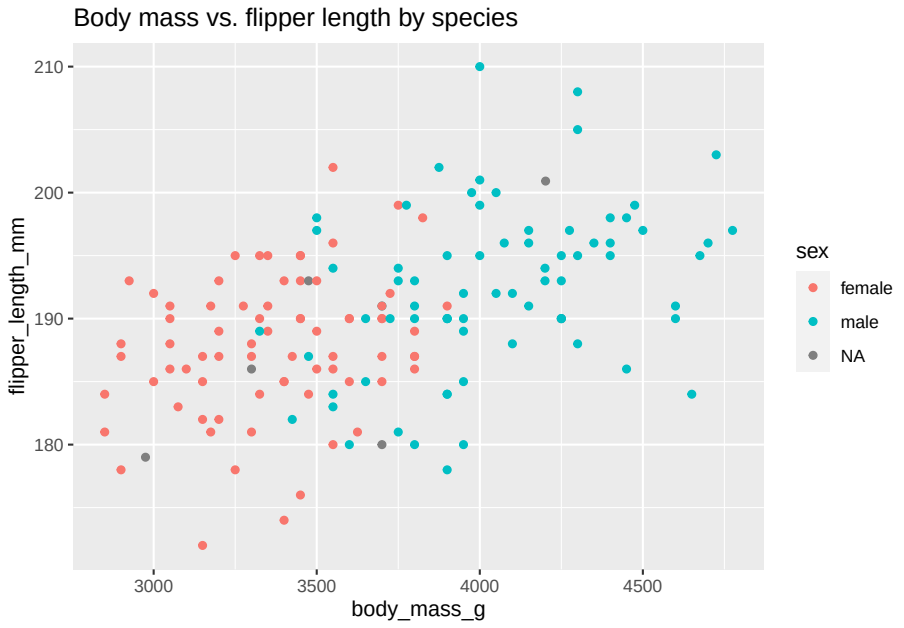
1. So assuming you have the data set called **data_imputed** made above where all the continuous measurements have been mean imputed you can address the missing sex problem for the *Adelie* species.

Maybe you can learn something about sexual dimorphism from these plot and table?

```
  table(is.na(data_imputed$sex),data_imputed$species)
```

```
##
##         Adelie Chinstrap Gentoo
##   FALSE    146        68    119
##   TRUE       6         0      5
```

```
  ggplot(data_imputed %>% filter(species %in% "Adelie"), aes(x = body_mass_g, y = flipp
  geom_point() +
  labs(title = "Body mass vs. flipper length by species")
```



Back to the future

Lets play a little more with grouping in the penguins dataset. Lets try to ignore the missing values for now instead of imputing them.

Get summary of a statistic, but grouping on another variable

```
penguins %>%
  group_by(species) %>%
  summarize(
    mean_bm =mean(body_mass_g, na.rm=T),
    sd_bm   =sd(body_mass_g, na.rm=T),
    n             = n()
)
```

```
## # A tibble: 3 x 4
##   species   mean_bm sd_bm     n
##   <fct>       <dbl> <dbl> <int>
## 1 Adelie      3701.  459.   152
## 2 Chinstrap   3733.  384.    68
```

```
## 3 Gentoo      5076.  504.   124
```

Grouping on more than one variable

```
penguins %>%
  group_by(species, sex) %>%
  summarize(
    mean_bm =mean(body_mass_g, na.rm=T),
    sd_bm   =sd(body_mass_g, na.rm=T),
    n             = n()
)
```

```
## `summarise()` has grouped output by 'species'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 8 x 5
## # Groups:   species [3]
##   species   sex    mean_bm sd_bm     n
##   <fct>     <fct>    <dbl> <dbl> <int>
## 1 Adelie    female   3369.  269.    73
## 2 Adelie    male     4043.  347.    73
## 3 Adelie    <NA>     3540   477.     6
## 4 Chinstrap female   3527.  285.    34
## 5 Chinstrap male     3939.  362.    34
## 6 Gentoo    female   4680.  282.    58
## 7 Gentoo    male     5485.  313.    61
## 8 Gentoo    <NA>     4588.  338.     5
```

Remove NAs sex variable and sort by mean body size

```
penguins %>%
  filter(!is.na(sex)) %>%
  group_by(species, sex) %>%
  summarize(
    mean_bm =mean(body_mass_g, na.rm=T),
    sd_bm   =sd(body_mass_g, na.rm=T),
    n             = n()
)%>%
arrange(desc(mean_bm))
```

```
## `summarise()` has grouped output by 'species'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 6 x 5
## # Groups:   species [3]
##   species   sex    mean_bm sd_bm     n
##   <fct>     <fct>    <dbl> <dbl> <int>
## 1 Gentoo    male     5485.  313.    61
## 2 Gentoo    female   4680.  282.    58
## 3 Adelie    male     4043.  347.    73
## 4 Chinstrap male     3939.  362.    34
## 5 Chinstrap female   3527.  285.    34
## 6 Adelie    female   3369.  269.    73
```

Great! That was the fast introduction to penguins. Think about what you learned from this. **The meta-information that you gather from doing exercises as the ones above is the real magic**. Therefore, as always, look at your data in different ways before diving in.

Lets look at a dataset from the internet

Open this link in a new tab by copy-pasting and save it to your computer using right-click (https://raw.githubusercontent.com/zief0002/miniature-garbanzo/main/data/gapminder.csv). Save it as "gapminder2017".

comma separated (csv)

try reading it in using read csv, the file ending is automatically put on there, so even though you did not name it gapminder2017.csv, that is the name it will get.

```
# read csv can create in comma-separated files, read.table can be used to read in tab separated t
# BUT in general, you just need to specify what separated your columns and if the data has a head
gapminder2017 <-read.csv("~/241023_phdcourse/data/gapminder2017.csv")

glimpse(gapminder2017)
```

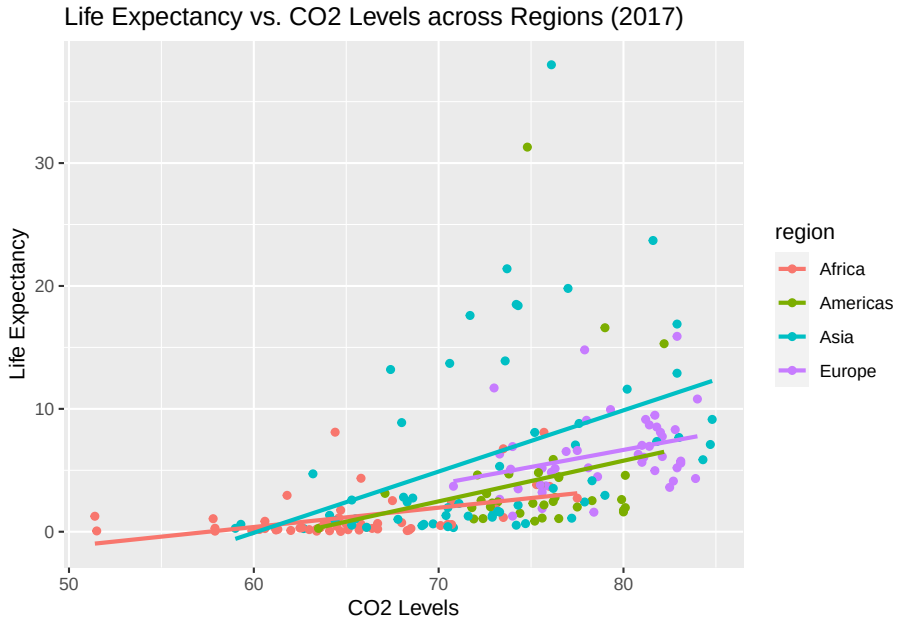
```
## Rows: 193
## Columns: 8
## $ country      <chr> "Afghanistan", "Albania", "Algeria", "Andorra", "Angola",~
## $ region       <chr> "Asia", "Europe", "Africa", "Europe", "Africa", "Americas~
## $ income       <dbl> 2.03, 13.30, 11.60, 58.30, 6.93, 21.00, 22.70, 12.70, 49.~
## $ income_level <chr> "Level 1", "Level 3", "Level 3", "Level 4", "Level 2", "L~
## $ life_exp     <dbl> 62.7, 78.4, 76.0, 82.1, 64.6, 76.2, 76.5, 75.6, 82.9, 82.~
## $ co2          <dbl> 0.254, 1.590, 3.690, 6.120, 1.120, 5.880, 4.410, 1.890, 1~
```

```
## $ co2_change   <chr> "increase", "increase", "increase", "decrease", "decrease~
## $ population   <dbl> 37.2000, 2.8800, 42.2000, 0.0770, 30.8000, 0.0963, 44.400~
```

Plotting is useful for understanding your data, in this case, we can plot the life expectancy and co2 across regions. We will back to plotting much more so dont worry if you dont understand it all right now.

```
ggplot(gapminder2017, aes(x = life_exp, y = co2, color = region)) +
  geom_point() +  # Add points for each data point
  geom_smooth(method = "lm", se = FALSE) +  # Add linear trendlines without confidence
  labs(title = "Life Expectancy vs. CO2 Levels across Regions (2017)", x = "CO2 Levels"
```

```
## `geom_smooth()` using formula = 'y ~ x'
```

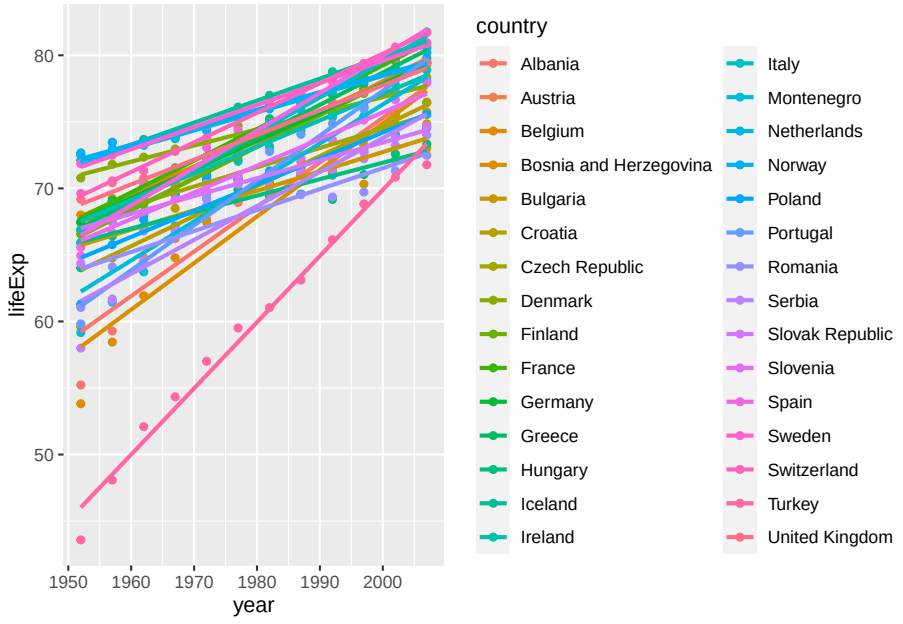


However, what we want is not a point estimate for the year 2017, we want the change in life expectancy over time across regions, but not per year. We want to do it per 5 years to remove the uncertainty given by yearly measurements also we only want to look at the Nordic countries. The full dataset for gapminder is available in the R package `gapminder` install this if you have not done so already.

For an overview of the data, look at this plot

```
library(gapminder)

# For Denmark see the trend in life expectancy
gapminder %>% filter(continent %in% "Europe") %>%
  ggplot(., aes(x = year, y = lifeExp, color = country)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)  # Add linear trendlines without confidence bands
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



Construct new date variable and look at life expectancy

Often, we are given raw data and we need to invent a new feature that incorporates the flaws of our data while still being useful. For example, yearly estimates might not be available and thus we want to contruct a 5 year period increase to remove some of uncertainty introduced by this effect.

```
# Filter data for Nordic countries in the 'gapminder' dataset
nordic_countries <- c("Denmark", "Finland", "Iceland", "Norway", "Sweden")
nordic_data <- gapminder %>% filter(country %in% nordic_countries)

# Create a date interval of five years
```

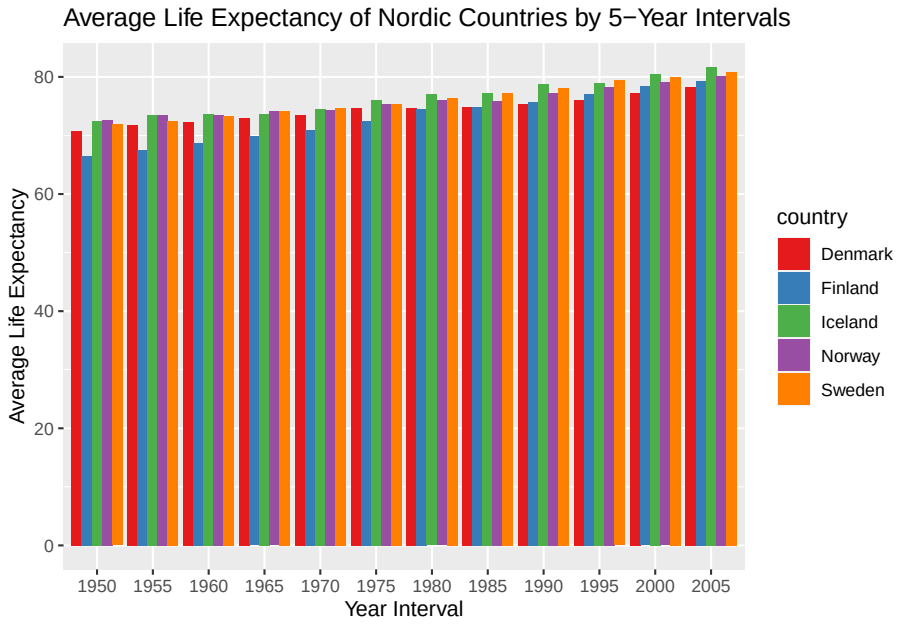
```
nordic_data <- nordic_data %>%
  mutate(year_interval = year %/% 5 * 5)  # Create intervals of 5 years

# Summarize life expectancy for the Nordic countries within the 5-year intervals
summary <- nordic_data %>%
  group_by(country, year_interval) %>%
  summarise(avg_life_expectancy = mean(lifeExp, na.rm = TRUE))
```

```
## `summarise()` has grouped output by 'country'. You can override using the
## `.groups` argument.
```

```
# Show the summarized data
summary
```

```
## # A tibble: 60 x 3
## # Groups:   country [5]
##    country year_interval avg_life_expectancy
##    <fct>           <dbl>               <dbl>
##  1 Denmark          1950                70.8
##  2 Denmark          1955                71.8
##  3 Denmark          1960                72.4
##  4 Denmark          1965                73.0
##  5 Denmark          1970                73.5
##  6 Denmark          1975                74.7
##  7 Denmark          1980                74.6
##  8 Denmark          1985                74.8
##  9 Denmark          1990                75.3
## 10 Denmark          1995                76.1
## # i 50 more rows
```

```
ggplot(summary, aes(x = factor(year_interval), y = avg_life_expectancy, fill = country)
  geom_bar(stat = "identity", position = "dodge") +
  labs(title = "Average Life Expectancy of Nordic Countries by 5-Year Intervals", x = "
  scale_fill_brewer(palette = "Set1")  # Adjust the color palette as desired
```



Your turn

Exercise 4.

Now using the dataset **summary** we want to see how the percentwise increase between 1950 and year 2005 differs between the Nordic countries

1. Create a new dataframe which contains the relative increase in life expectancy across Nordic countries between years and arrange from highest to lowest