# Lecture Notes for the Tidy R Seminar

CED, Barcelona, Summer 2018.

*Jonas Schöley*

*2018-07-16*

# Contents

# Chapter 1

# Before we start

## 1.1 Prerequisites

Please run this chunk of code below in your R session. It will install all the packages we need during the course.

```
install.packages('devtools')

devtools::install_cran(
  pkgs = c(
    'tidyverse',      # tools for tidy data analysis
    'rmarkdown',      # literate programming
    'plotly',         # interactive visualization
    'ggmap',          # use online map-tiles with ggplot
    'eurostat',       # download data from eurostat
    'sf',             # tidy geo-computing
    'rnaturalearth', # download worldwide map data
    'gapminder',      # data from the gapminder world project
    'cowplot',        # ggplot multiple figures addon
    'skimr'           # nice dataframe summaries
  ),
  repos = "https://cran.rstudio.com/"
)
```

## 1.2 Literate programming in Rmarkdown

The official Rmarkdown cheat sheet.

### 1.2.1 Excercise: Literate programming in Rmarkdown

- Rewrite any of your existing R-scripts into an Rmarkdown script. Demonstrate loading data from the hard disc, plot output, text blocks, headers and lists. Export to pdf.

# Chapter 2

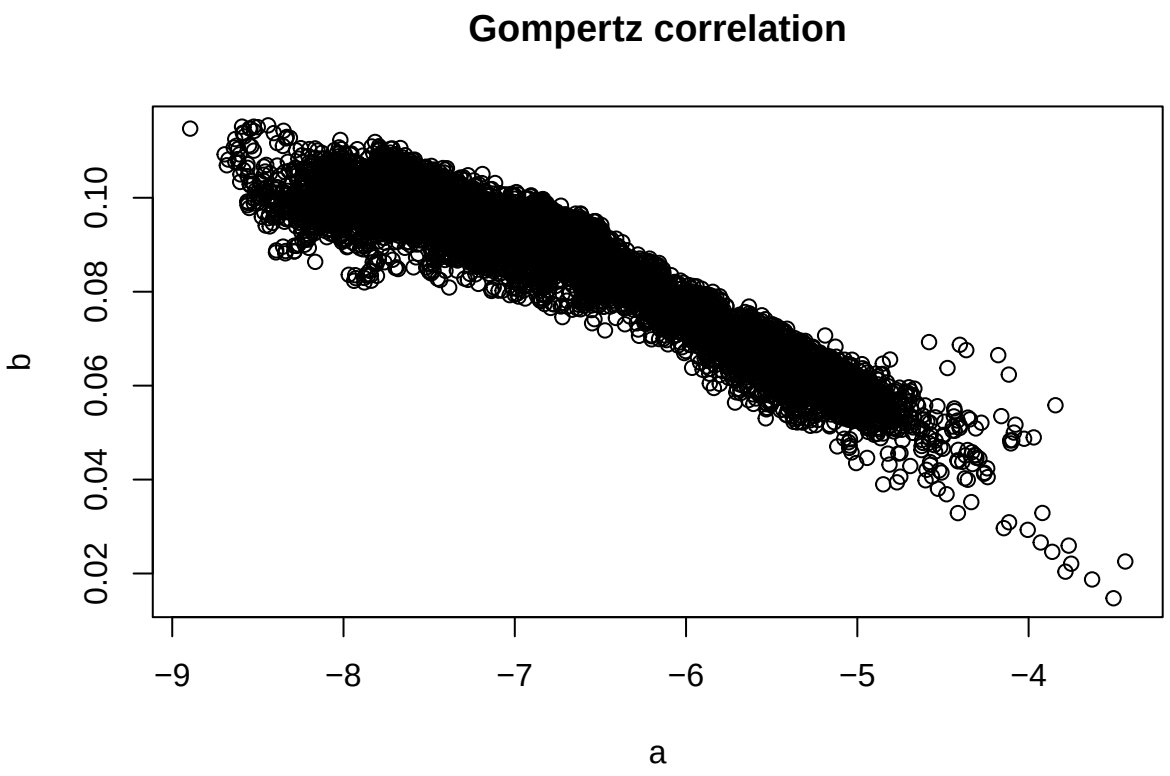
# What is tidy R?

## 2.1 tidy R vs. base R

The Gompertz Law describes the exponential relationship between age and mortality rate in human populations. Its two parameters $a$ and $b$ describe the overall mortality level and the relative rate of mortality inrease over age. It is well known that both parameters correlate with each other. The code below demonstrates this co-variance by fitting the Gompertz equation to Swedish birth cohorts and plotting the parameter estimates against each other.

$$\mu(x) = ae^{bx}$$

```
# load data
load('data/hmd/hmd_counts.RData')

# select ages 30 to 80, drop total counts, drop NAs
hmd_sub <-
  na.omit(subset(hmd_counts, age >= 30 & age < 80 & sex != 'Total'))
# split the data by sex, country and year
hmd_split <-
  split(hmd_sub, list(hmd_sub$sex, hmd_sub$country, hmd_sub$period),
        drop = TRUE)
# run a linear regression on each subset
hmd_regress <-
  lapply(hmd_split,
         function (lt) glm(round(nDx, 0) ~ I(age-30) + offset(log(nEx)),
                           family = 'poisson', data = lt))
# extract the coefficients from each regression model
hmd_coef <- t(sapply(hmd_regress, coef))
# plot a versus b coefficients
plot(x = hmd_coef[,1], y = hmd_coef[,2],
     main = 'Gompertz correlation', xlab = 'a', ylab = 'b')
```

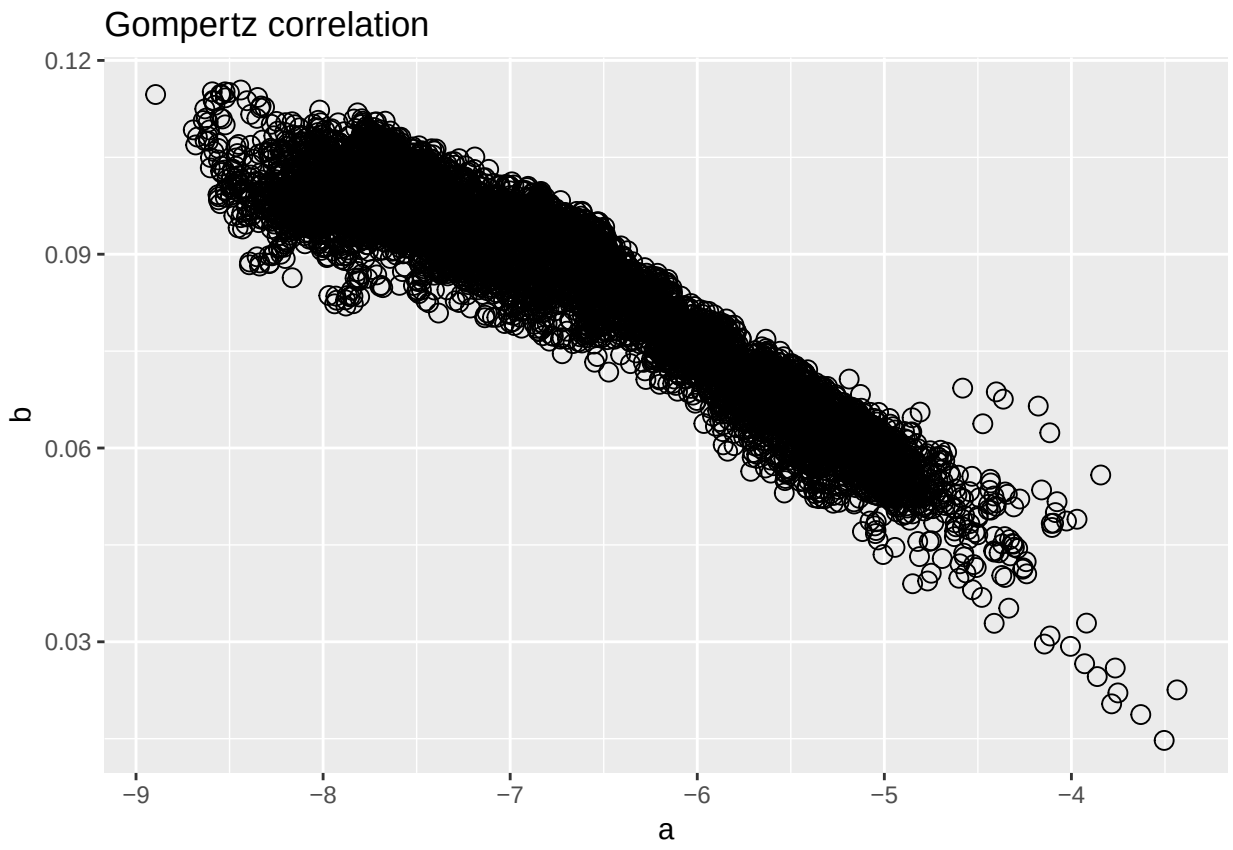



Here's how I would write the same program today, using the `tidyverse`.

```
#library(tidyverse)
# load data
load('data/hmd/hmd_counts.RData')

hmd_counts %>%
  # select ages 30 to 80, drop total counts
  filter(age >= 30, age < 80, sex != 'Total') %>%
  # drop NAs
  drop_na() %>%
  # for each period...
  group_by(period, country, sex) %>%
  # ...run a Poisson regression of deaths versus age
  do(lm = glm(round(nDx, 0) ~ I(age-30) + offset(log(nEx)),
              family = 'poisson', data = .)) %>%
  # extract the regression coefficients
  mutate(a = coef(lm)[1], b = coef(lm)[2]) %>%
  # plot a versus b coefficients and label with year
  ggplot() +
  geom_point(aes(x = a, y = b), shape = 1, size = 3) +
  labs(title = 'Gompertz correlation')
```

```
# load data
load('data/hmd/hmd_counts.RData')

# select ages 30 to 80, drop total counts, drop NAs
hmd_sub <-
  na.omit(subset(hmd_counts, age >= 30 & age < 80 & sex != 'Total'))
# split the data by sex, country and year
hmd_split <-
  split(hmd_sub, list(hmd_sub$sex, hmd_sub$country, hmd_sub$period),
        drop = TRUE)
# run a linear regression on each subset
hmd_regress <-
  lapply(hmd_split,
         function (lt) glm(round(nDx, 0) ~ I(age-30) + offset(log(nEx)),
                           family = 'poisson', data = lt))
# extract the coefficients from each regression model
hmd_coef <- t(sapply(hmd_regress, coef))
# plot a versus b coefficients
plot(x = hmd_coef[,1], y = hmd_coef[,2],
     main = 'Gompertz correlation', xlab = 'a', ylab = 'b')
```

Figure 2.1: In base R we stores intermediate results often.




What are the differences?

```
library(tidyverse)
# load data
load('data/hmd/hmd_counts.RData')

hmd_counts %>%
  # select ages 30 to 80, drop total counts
  filter(age >= 30, age < 80, sex != 'Total') %>%
  # drop NAs
  drop_na() %>%
  # for each period...
  group_by(period, country, sex) %>%
  # ...run a Poisson regression of deaths versus age
  do(lm = glm(round(nDx, 0) ~ I(age-30) + offset(log(nEx)),
              family = 'poisson', data = .)) %>%
  # extract the regression coefficients
  mutate(a = coef(lm)[1], b = coef(lm)[2]) %>%
  # plot a versus b coefficients and label with year
  ggplot() +
  geom_point(aes(x = a, y = b), shape = 1, size = 3) +
  labs(title = 'Gompertz correlation')
```

Figure 2.2: In tidy R a single *data pipeline* often gets us from raw data to finished result.

```
# load data
load('data/hmd/hmd_counts.RData')

# select ages 30 to 80, drop total counts, drop NAs
hmd_sub <-
  na.omit(subset(hmd_counts, age >= 30 & age < 80 & sex != 'Total'))
# split the data by sex, country and year
hmd_split <-
  split(hmd_sub, list(hmd_sub$sex, hmd_sub$country, hmd_sub$period),
        drop = TRUE)
# run a linear regression on each subset
hmd_regress <-
  lapply(hmd_split,
         function (lt) glm(round(nDx, 0) ~ I(age-30) + offset(log(nEx)),
                           family = 'poisson', data = lt))
# extract the coefficients from each regression model
hmd_coef <- t(sapply(hmd_regress, coef))
# plot a versus b coefficients
plot(x = hmd_coef[,1], y = hmd_coef[,2],
     main = 'Gompertz correlation', xlab = 'a', ylab = 'b')
```

Dataframe
List
Matrix

Figure 2.3: In base R data structures change often.

Dataframe

```
library(tidyverse)
# load data
load('data/hmd/hmd_counts.RData')

hmd_counts %>%
  # select ages 30 to 80, drop total counts
  filter(age >= 30, age < 80, sex != 'Total') %>%
  # drop NAs
  drop_na() %>%
  # for each period...
  group_by(period, country, sex) %>%
  # ...run a Poisson regression of deaths versus age
  do(lm = glm(round(nDx, 0) ~ I(age-30) + offset(log(nEx)),
              family = 'poisson', data = .)) %>%
  # extract the regression coefficients
  mutate(a = coef(lm)[1], b = coef(lm)[2]) %>%
  # plot a versus b coefficients and label with year
  ggplot() +
  geom_point(aes(x = a, y = b), shape = 1, size = 3) +
  labs(title = 'Gompertz correlation')
```

Figure 2.4: In tidy R the dataframe is the most important data structure.

NSE
List index
Matrix index

```
# load data
load('data/hmd/hmd_counts.RData')

# select ages 30 to 80, drop total counts, drop NAs
hmd_sub <-
  na.omit(subset(hmd_counts, age >= 30 & age < 80 & sex != 'Total'))
# split the data by sex, country and year
hmd_split <-
  split(hmd_sub, list(hmd_sub$sex, hmd_sub$country, hmd_sub$period),
        drop = TRUE)
# run a linear regression on each subset
hmd_regress <-
  lapply(hmd_split,
         function (lt) glm(round(nDx, 0) ~ I(age-30) + offset(log(nEx)),
                           family = 'poisson', data = lt))
# extract the coefficients from each regression model
hmd_coef <- t(sapply(hmd_regress, coef))
# plot a versus b coefficients
plot(x = hmd_coef[,1], y = hmd_coef[,2],
     main = 'Gompertz correlation', xlab = 'a', ylab = 'b')
```

Figure 2.5: In base R we have to use various indexing styles.

NSE

```
library(tidyverse)
# load data
load('data/hmd/hmd_counts.RData')

hmd_counts %>%
  # select ages 30 to 80, drop total counts
  filter(age >= 30, age < 80, sex != 'Total') %>%
  # drop NAs
  drop_na() %>%
  # for each period...
  group_by(period, country, sex) %>%
  # ...run a Poisson regression of deaths versus age
  do(lm = glm(round(nDx, 0) ~ I(age-30) + offset(log(nEx)),
              family = 'poisson', data = .)) %>%
  # extract the regression coefficients
  mutate(a = coef(lm)[1], b = coef(lm)[2]) %>%
  # plot a versus b coefficients and label with year
  ggplot() +
  geom_point(aes(x = a, y = b), shape = 1, size = 3) +
  labs(title = 'Gompertz correlation')
```

Figure 2.6: In tidy R we use a single indexing style.

```
# load data
load('data/hmd/hmd_counts.RData')

# select ages 30 to 80, drop total counts, drop NAs
hmd_sub <-
  na.omit(subset(hmd_counts, age >= 30 & age < 80 & sex != 'Total'))
# split the data by sex, country and year
hmd_split <-
  split(hmd_sub, list(hmd_sub$sex, hmd_sub$country, hmd_sub$period),
        drop = TRUE)
# run a
hmd_regre
  lapply(
                                                    ge-30) + offset(log(nEx)),
                                                     data = lt))
# extract
hmd_coef                                            n model
# plot a versus b coefficients
plot(x = hmd_coef[,1], y = hmd_coef[,2],
     main = 'Gompertz correlation', xlab = 'a', ylab = 'b')
```

```
                 (Intercept) I(age - 30)
Female.SWE.1751   -4.954454  0.05379309
Male.SWE.1751     -4.834564  0.05463634
Female.SWE.1752   -5.109355  0.05332976
Male.SWE.1752     -4.973304  0.05313042
Female.SWE.1753   -5.176551  0.05545279
Male.SWE.1753     -4.946853  0.05258433
```

Figure 2.7: In base R important information is sometimes stored in row names.

```
library(tidyverse)
# load data
load('data/hmd/hmd_counts.RData')

hmd_counts %>%
  # select ages 30 to 80, drop total
  filter(age >= 30, age < 80, sex != 
  # drop NAs
  drop_na() %>%
  # for each period...
  group_by(period, country, sex) %>%
  # ...run a Poisson regression of d
  do(lm = glm(round(nDx, 0) ~ I(age-
              family = 'poisson', da
  # extract the regression coefficie
  mutate(a = coef(lm)[1], b = coef(lm)[2]) %>%
  # plot a versus b coefficients and label with year
  ggplot() +
  geom_point(aes(x = a, y = b), shape = 1, size = 3) +
  labs(title = 'Gompertz correlation')
```

```
# A tibble: 7,826 x 6
   period country sex    lm            a      b
    <int> <chr>   <chr>  <list>    <dbl>  <dbl>
 1   1751 SWE     Female <S3: glm> -4.95 0.0538
 2   1751 SWE     Male   <S3: glm> -4.83 0.0546
 3   1752 SWE     Female <S3: glm> -5.11 0.0533
 4   1752 SWE     Male   <S3: glm> -4.97 0.0531
 5   1753 SWE     Female <S3: glm> -5.18 0.0555
 6   1753 SWE     Male   <S3: glm> -4.95 0.0526
 7   1754 SWE     Female <S3: glm> -5.11 0.0561
 8   1754 SWE     Male   <S3: glm> -4.82 0.0528
 9   1755 SWE     Female <S3: glm> -5.03 0.0551
10   1755 SWE     Male   <S3: glm> -4.82 0.0523
# ... with 7,816 more rows
```

Figure 2.8: In tidy R all information exists in the form of data frame columns.

### 2.1.1 Excercise: tidy R vs. base R

- Discuss the differences.

## 2.2 First steps in tidy R

This is the script we developed during the first lecture.

Everything starts with a dataframe.

```
#library(tidyverse)

load('data/hmd/hmd_counts.RData')

hmd_counts %>%
  arrange(country, sex, period, age)
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx   nDx    nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     Female   1921     0     1 3842. 64052.
## 2 AUS     Female   1921     1     1  719. 59619.
## 3 AUS     Female   1921     2     1  330. 57126.
## 4 AUS     Female   1921     3     1  166. 57484.
## 5 AUS     Female   1921     4     1  190. 58407.
## # ... with 1.305e+06 more rows
```

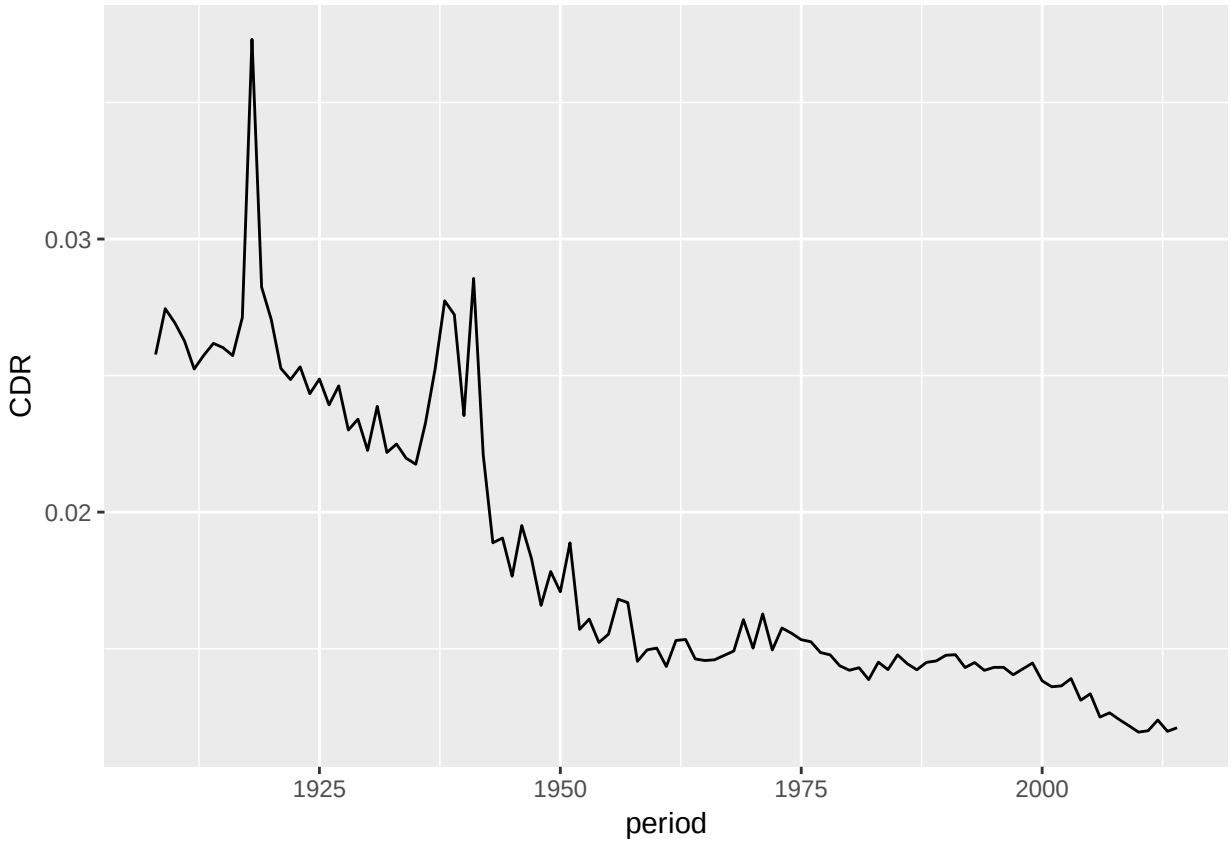
*Split-apply-combine* is such a common data analysis pattern.

`summarise()` can work with base R functions that take a vector as input and return a scalar (a single value) as output. If you work with a function that returns a small vector (like `range()`) you must make sure to subset the output of that function to a scalar.

```
counts <-
  hmd_counts %>%
  arrange(country, sex, period, age) %>%
  filter(age >= 30, sex == 'Total') %>%
  drop_na() %>%
  group_by(country) %>%
  summarise(min = range(period)[1],
            max = range(period)[2])
```

We can write a pipe end-to-end, from raw data, to finished plot, without ever saving the results. It is important to know how your data looks like at each stage in the pipe. You can write `data %>% View()` to get that information.

```
hmd_counts %>%
  arrange(country, sex, period, age) %>%
  filter(age >= 30, sex == 'Total') %>%
  drop_na() %>%
  group_by(country, period) %>%
  summarise(D = sum(nDx), E = sum(nEx)) %>%
  mutate(CDR = D/E) %>%
  filter(country == 'ESP') %>%
  ggplot() +
  geom_line(aes(x = period, y = CDR))
```



# Chapter 3

# Data Wrangling

## 3.1 Datasets used in this chapter

```
library(tidyverse)

# hmd_counts
# Deaths and exposures by age, year, sex and country
# source: Human Mortality Database
load('data/hmd/hmd_counts.RData')

# General social survey
# source: http://gss.norc.org/s
gss <- haven::read_stata('data/gss/GSS2016.DTA')

# euro_regio
# European regional population statistics
load('data/euro_regio/euro_regio.Rdata')

# eu_timeuse_tot
# European timeuse survey
# source: Eurostat
load('data/eu_timeuse/eu_timeuse_tot.Rdata')

# hmd
# Life tables by year, sex and country
# source: Human Mortality Database
load('data/hmd/hmd.RData')
```

## 3.2 The "tidy" approach to data wrangling

"Data wrangling" is the process of transforming raw data into a form fit for analysis. This may include operations like filtering, sorting, joining, splitting, recoding or reshaping, nearly always performed in conjuction with each other. The tidy approach to data wrangling aims to make this often mundane but critical task as clear and fast as possible. Tidy data wrangling revolves around 4 concepts:

1) **Verbs** A small collection of "verbs" provides basic transformation operations.

2) **Pipes** Transformations can be chained together via a "pipe".
3) **Tidy Data** Everything is a data frame with cases as rows and variables as columns.
4) **Tidy iteration** Transformations can be repeatedly applied to different row or column subsets without the use of a for loop.

## 3.3 6 verbs for data transformation

The `tidyverse` provides us with an array of *verbs* for data transformation such as `mutate()`, `filter()`, `arrange()` and others. While each of those verbs performs a distinct action they all have a common design. All verbs...

- ...have as their first argument a data frame:

All verbs *operate on data frames only*. This is one of the helpful restrictions of the tidyverse because you don't need to consider data structures beyond the tabular form of a data frame. If your data does not come in a data frame, it needs to be converted.

```
# this fails because "WorldPhones" is a matrix and not a data frame
select(WorldPhones, 3)
```

```
## Error in UseMethod("select_"): no applicable method for 'select_' applied to an object of class "c('m
```

```
# converting World Phones to a data frame resolves the issue
select(as.data.frame(WorldPhones), 3)
```

```
##      Asia
## 1951 2876
## 1956 4708
## 1957 5230
## 1958 6662
## 1959 6856
## 1960 8220
## 1961 9053
```

The data frame to operate on is always the first argument in any of the *verb* functions. Here we `select()` the first variable of the data frame `hmd_counts`.

```
select(hmd_counts, 1)
```

```
## # A tibble: 1,304,694 x 1
##   country
##   <chr>
## 1 AUS
## 2 AUS
## 3 AUS
## 4 AUS
## 5 AUS
## # ... with 1.305e+06 more rows
```

Due to the data frame always coming first we can use the `%>%` (pipe) operator to pass data into any of the verbs.

```
hmd_counts %>% select(1)
```

```
## # A tibble: 1,304,694 x 1
##   country
##   <chr>
## 1 AUS
```

```
## 2 AUS
## 3 AUS
## 4 AUS
## 5 AUS
## # ... with 1.305e+06 more rows
```

- …don't change the data frame unless you explicitly want to

Note that none of the verbs permanently change the content of a data frame. You need to assign the output of a verb to an object in order to permanently store your computations.

```
# display the result of rename() without changing the data
rename(hmd_counts, deaths = nDx, exposures = nEx)
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx deaths exposures
##   <chr>   <chr>   <int> <int> <int>  <dbl>     <dbl>
## 1 AUS     Female   1921     0     1  3842.    64052.
## 2 AUS     Female   1921     1     1   719.    59619.
## 3 AUS     Female   1921     2     1   330.    57126.
## 4 AUS     Female   1921     3     1   166.    57484.
## 5 AUS     Female   1921     4     1   190.    58407.
## # ... with 1.305e+06 more rows
```

```
hmd_counts
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx   nDx    nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     Female   1921     0     1 3842. 64052.
## 2 AUS     Female   1921     1     1  719. 59619.
## 3 AUS     Female   1921     2     1  330. 57126.
## 4 AUS     Female   1921     3     1  166. 57484.
## 5 AUS     Female   1921     4     1  190. 58407.
## # ... with 1.305e+06 more rows
```

```
# permanently store the results of rename() in a new object called `hmd_counts_new`
hmd_counts_new <- rename(hmd_counts, deaths = nDx, exposures = nEx)
hmd_counts_new
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx deaths exposures
##   <chr>   <chr>   <int> <int> <int>  <dbl>     <dbl>
## 1 AUS     Female   1921     0     1  3842.    64052.
## 2 AUS     Female   1921     1     1   719.    59619.
## 3 AUS     Female   1921     2     1   330.    57126.
## 4 AUS     Female   1921     3     1   166.    57484.
## 5 AUS     Female   1921     4     1   190.    58407.
## # ... with 1.305e+06 more rows
```

- …let you address columns within the data frame by simply typing the name

The tidyverse functions always work within the context of a data frame – specified as first argument – and columns within that data frame can be adressed by simply typing their name (without quotes).

```
# this returns life-tables entries for Australia, 1921 ages 20 to 25
filter(hmd_counts, country == 'AUS', period == 1921, age %in% 20:25)
```

```
## # A tibble: 18 x 7
```

```
##   country sex    period   age    nx   nDx    nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     Female   1921    20     1  138. 46387.
## 2 AUS     Female   1921    21     1  118. 46437.
## 3 AUS     Female   1921    22     1  116. 45769.
## 4 AUS     Female   1921    23     1  149. 45746.
## 5 AUS     Female   1921    24     1  146. 46524.
## # ... with 13 more rows
```

```
# this is the same, but much more cumbersome to write and read
hmd_counts[hmd_counts$country == 'AUS' &
             hmd_counts$period == 1921 &
             hmd_counts$age %in% 20:25,]
```

```
## # A tibble: 18 x 7
##   country sex    period   age    nx   nDx    nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     Female   1921    20     1  138. 46387.
## 2 AUS     Female   1921    21     1  118. 46437.
## 3 AUS     Female   1921    22     1  116. 45769.
## 4 AUS     Female   1921    23     1  149. 45746.
## 5 AUS     Female   1921    24     1  146. 46524.
## # ... with 13 more rows
```

- ...output a data frame

### 3.3.1 Column based transforms

#### 3.3.1.1 `mutate()` columns

`mutate()` adds a column to a data frame or changes an existing column.

```
hmd_counts %>%
  # calculate new column with death rates from
  # columns nDx and nEx
  mutate(nmx = nDx/nEx)
```

```
## # A tibble: 1,304,694 x 8
##   country sex    period   age    nx   nDx    nEx     nmx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>   <dbl>
## 1 AUS     Female   1921     0     1 3842. 64052. 0.0600
## 2 AUS     Female   1921     1     1  719. 59619. 0.0121
## 3 AUS     Female   1921     2     1  330. 57126. 0.00578
## 4 AUS     Female   1921     3     1  166. 57484. 0.00289
## 5 AUS     Female   1921     4     1  190. 58407. 0.00325
## # ... with 1.305e+06 more rows
```

```
hmd_counts %>%
  # change unit of exposure variable from person-years
  # to person months
  mutate(nEx = nEx*12)
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx   nDx     nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>   <dbl>
## 1 AUS     Female   1921     0     1 3842. 768626.
```

```
## 2 AUS     Female   1921     1     1  719. 715431.
## 3 AUS     Female   1921     2     1  330. 685512.
## 4 AUS     Female   1921     3     1  166. 689806.
## 5 AUS     Female   1921     4     1  190. 700884.
## # ... with 1.305e+06 more rows
```

Within a single `mutate()` statement multiple new variables can be created.

```
hmd_counts %>%
  mutate(
    # add discrete age variable
    age_group = cut(age, seq(0, 90, 10), include.lowest = TRUE),
    # add mortality rate
    nmx = nDx/nEx
  )
```

```
## # A tibble: 1,304,694 x 9
##   country sex    period   age    nx   nDx    nEx age_group     nmx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl> <fct>       <dbl>
## 1 AUS     Female   1921     0     1 3842. 64052. [0,10]    0.0600
## 2 AUS     Female   1921     1     1  719. 59619. [0,10]    0.0121
## 3 AUS     Female   1921     2     1  330. 57126. [0,10]    0.00578
## 4 AUS     Female   1921     3     1  166. 57484. [0,10]    0.00289
## 5 AUS     Female   1921     4     1  190. 58407. [0,10]    0.00325
## # ... with 1.305e+06 more rows
```

Newly created variables can immediately be used in the creation of additional variables.

```
hmd_counts %>%
  mutate(
    nmx = nDx/nEx,
    # convert death rates to death probability
    nqx = 1-exp(-nmx),
    # and survival probability
    npx = 1-nqx
  )
```

```
## # A tibble: 1,304,694 x 10
##   country sex    period   age    nx   nDx    nEx     nmx     nqx   npx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>   <dbl>   <dbl> <dbl>
## 1 AUS     Female   1921     0     1 3842. 64052. 0.0600  0.0582  0.942
## 2 AUS     Female   1921     1     1  719. 59619. 0.0121  0.0120  0.988
## 3 AUS     Female   1921     2     1  330. 57126. 0.00578 0.00576 0.994
## 4 AUS     Female   1921     3     1  166. 57484. 0.00289 0.00288 0.997
## 5 AUS     Female   1921     4     1  190. 58407. 0.00325 0.00325 0.997
## # ... with 1.305e+06 more rows
```

#### 3.3.1.2 `select()` columns

Using `select()` we can specify which columns to keep and which columns to delete from a data frame. Let's have a look at a typical panel data set: The General Social Survey. We can see that the `gss` data features 960 columns.

```
gss
```

```
## # A tibble: 2,867 x 960
##   mar1   mar2  mar3  mar4  mar5  mar6  mar7  mar8  mar9  mar10 mar11 mar12
```

```
##   <dbl+> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1      1     5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 2 4      <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 3 1      1     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 4 1      1     5     4     5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 5 1      1     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## # ... with 2,862 more rows, and 948 more variables: mar13 <dbl+lbl>,
## #   mar14 <dbl+lbl>, abany <dbl+lbl>, abdefect <dbl+lbl>,
## #   abhlth <dbl+lbl>, ...
```

Here we just select the two columns named `id` and `age`.

```
gss %>% select(id, age)
```

```
## # A tibble: 2,867 x 2
##      id age
##   <dbl> <dbl+lbl>
## 1     1 47
## 2     2 61
## 3     3 72
## 4     4 43
## 5     5 55
## # ... with 2,862 more rows
```

We can also select by column position.

```
gss %>% select(312, 28)
```

```
## # A tibble: 2,867 x 2
##      id age
##   <dbl> <dbl+lbl>
## 1     1 47
## 2     2 61
## 3     3 72
## 4     4 43
## 5     5 55
## # ... with 2,862 more rows
```

The column operator `:` selects a range of columns. We can use it to select all variables from `mar1` to `mar14`, the marriage status of up to 14 persons in a household.

```
gss %>% select(mar1:mar14)
```

```
## # A tibble: 2,867 x 14
##   mar1   mar2  mar3  mar4  mar5  mar6  mar7  mar8  mar9  mar10 mar11 mar12
##   <dbl+> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1      1     5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 2 4      <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 3 1      1     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 4 1      1     5     4     5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 5 1      1     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## # ... with 2,862 more rows, and 2 more variables: mar13 <dbl+lbl>,
## #   mar14 <dbl+lbl>
```

Again, the same is possible by specifying the column position, in this case the first 14 columns.

```
gss %>% select(1:14)
```

```
## # A tibble: 2,867 x 14
```

```
##   mar1   mar2  mar3  mar4  mar5  mar6  mar7  mar8  mar9  mar10 mar11 mar12
##   <dbl+> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1      1     5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 2 4      <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 3 1      1     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 4 1      1     5     4     5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 5 1      1     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## # ... with 2,862 more rows, and 2 more variables: mar13 <dbl+lbl>,
## #   mar14 <dbl+lbl>
```

A minus preceeding a selection returns all columns *apart* from those specified. Notice that a selection of columns like `mar1:mar14` must be surrounded by parantheses in order to be removed.

```
# select everything apart from mar1
gss %>% select(-mar1)
```

```
## # A tibble: 2,867 x 959
##   mar2   mar3  mar4  mar5  mar6  mar7  mar8  mar9  mar10 mar11 mar12 mar13
##   <dbl+> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1      5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 2 <NA>   <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 3 1      <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 4 1      5     4     5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 5 1      5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## # ... with 2,862 more rows, and 947 more variables: mar14 <dbl+lbl>,
## #   abany <dbl+lbl>, abdefect <dbl+lbl>, abhlth <dbl+lbl>,
## #   abnomore <dbl+lbl>, ...
```

```
# select everything apart from mar1 to mar14
gss %>% select(-(mar1:mar14))
```

```
## # A tibble: 2,867 x 946
##   abany  abdefect abhlth abnomore abpoor abrape absingle acqntsex adforjob
##   <dbl+> <dbl+lb> <dbl+> <dbl+lb> <dbl+> <dbl+> <dbl+lb> <dbl+lb> <dbl+lb>
## 1 1      1        1      1        1      1      1        <NA>     <NA>
## 2 <NA>   <NA>     <NA>   <NA>     <NA>   <NA>   <NA>     2        <NA>
## 3 2      2        1      2        2      1      2        <NA>     <NA>
## 4 2      2        1      2        2      1      2        <NA>     <NA>
## 5 1      1        1      1        1      1      1        <NA>     <NA>
## # ... with 2,862 more rows, and 937 more variables: adults <dbl+lbl>,
## #   advfront <dbl+lbl>, advsched <dbl+lbl>, affrmact <dbl+lbl>,
## #   age <dbl+lbl>, ...
```

You can use functions inside of `select()` as long as they return either a column name or a column position. The standard R function `which()` is handy in that context as it returns the index of the elements for which a conditions holds true.

```
# return all columns of gss which names are 3 characters or less
gss %>% select(which(str_length(names(.)) <= 3))
```

```
## # A tibble: 2,867 x 6
##   age       god           id jew       sex       tax
##   <dbl+lbl> <dbl+lbl> <dbl> <dbl+lbl> <dbl+lbl> <dbl+lbl>
## 1 47        2             1 <NA>      1         2
## 2 61        <NA>          2 <NA>      1         <NA>
## 3 72        6             3 <NA>      1         1
## 4 43        6             4 <NA>      2         1
```

```
## 5 55        1             5 <NA>      2         1
## # ... with 2,862 more rows
```

There are a number of functions provided by the `tidyverse` which are designed to help with column selection.

```
# select all columns where name starts with 'age'
gss %>% select(starts_with('age'))
```

```
## # A tibble: 2,867 x 6
##   age       age3      aged      agedchld  agedpar   agekdbrn
##   <dbl+lbl> <dbl+lbl> <dbl+lbl> <dbl+lbl> <dbl+lbl> <dbl+lbl>
## 1 47        <NA>      <NA>      <NA>      3         29
## 2 61        <NA>      1         <NA>      <NA>      <NA>
## 3 72        <NA>      1         <NA>      <NA>      24
## 4 43        <NA>      <NA>      <NA>      1         19
## 5 55        <NA>      1         <NA>      <NA>      31
## # ... with 2,862 more rows
```

```
# select columns named mar1, mar2, mar3, mar4, mar5
gss %>% select(num_range('mar', 1:5))
```

```
## # A tibble: 2,867 x 5
##   mar1      mar2      mar3      mar4      mar5
##   <dbl+lbl> <dbl+lbl> <dbl+lbl> <dbl+lbl> <dbl+lbl>
## 1 1         1         5         5         <NA>
## 2 4         <NA>      <NA>      <NA>      <NA>
## 3 1         1         <NA>      <NA>      <NA>
## 4 1         1         5         4         5
## 5 1         1         5         <NA>      <NA>
## # ... with 2,862 more rows
```

```
# select all columns where name contains 'sex'
gss %>% select(contains('sex'))
```

```
## # A tibble: 2,867 x 17
##   acqntsex  frndsex  homosex  intsex   matesex  othersex paidsex  pikupsex
##   <dbl+lbl> <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb>
## 1 <NA>      <NA>     4        2        <NA>     <NA>     <NA>     <NA>
## 2 2         1        <NA>     2        2        2        2        2
## 3 <NA>      <NA>     4        2        1        <NA>     <NA>     <NA>
## 4 <NA>      <NA>     4        2        <NA>     <NA>     <NA>     <NA>
## 5 <NA>      <NA>     4        2        1        <NA>     <NA>     <NA>
## # ... with 2,862 more rows, and 9 more variables: relatsex <dbl+lbl>,
## #   sex <dbl+lbl>, sexeduc <dbl+lbl>, sexfreq <dbl+lbl>,
## #   sexornt <dbl+lbl>, ...
```

You can also use regular expressions when selecting columns.

```
# select columns where the name contains a number
gss %>% select(matches('[0-9]'))
```

```
## # A tibble: 2,867 x 323
##   mar1   mar2  mar3  mar4  mar5  mar6  mar7  mar8  mar9  mar10 mar11 mar12
##   <dbl+> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1      1     5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 2 4      <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 3 1      1     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## 4 1      1     5     4     5     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
```

```
## 5 1      1     5     <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
## # ... with 2,862 more rows, and 311 more variables: mar13 <dbl+lbl>,
## #   mar14 <dbl+lbl>, age3 <dbl+lbl>, artatt1 <dbl+lbl>, artatt2 <dbl+lbl>,
## #   ...
```

`select()` returns the columns in the order it selects them. It can therefore be used to reorder the columns of a dataframe.

```
# reverse the order of the columns
gss %>% select(rev(names(.)))
```

```
## # A tibble: 2,867 x 960
##   cogradtounder cobarate  coeftotlt cosector  spgradtounder spbarate
##   <dbl+lbl>     <dbl+lbl> <dbl+lbl> <dbl+lbl> <dbl+lbl>     <dbl+lbl>
## 1 <NA>          <NA>      <NA>      <NA>      2             3
## 2 <NA>          <NA>      <NA>      <NA>      <NA>          <NA>
## 3 <NA>          <NA>      <NA>      <NA>      <NA>          <NA>
## 4 <NA>          <NA>      <NA>      <NA>      <NA>          <NA>
## 5 <NA>          <NA>      <NA>      <NA>      1             1
## # ... with 2,862 more rows, and 954 more variables: speftotlt <dbl+lbl>,
## #   spsector <dbl+lbl>, gradtounder <dbl+lbl>, barate <dbl+lbl>,
## #   eftotlt <dbl+lbl>, ...
```

```
# reorder the columns in reverse alphabetical order
gss %>% select(order(names(.), decreasing = TRUE))
```

```
## # A tibble: 2,867 x 960
##   zodiac  year xnorcsiz xmovie xmarsex xhaustn wwwmin wwwhr wtssnr wtssall
##   <dbl+> <dbl> <dbl+lb> <dbl+> <dbl+l> <dbl+l> <dbl+> <dbl> <dbl+> <dbl+l>
## 1 11      2016 6        <NA>   2       <NA>    0      15    1.260~ 0.9569~
## 2 8       2016 6        2      <NA>    <NA>    0      5     0.630~ 0.4784~
## 3 12      2016 6        2      1       <NA>    <NA>   <NA>  1.260~ 0.9569~
## 4 4       2016 6        <NA>   1       <NA>    0      7     2.520~ 1.9139~
## 5 8       2016 6        2      1       3       <NA>   <NA>  1.890~ 1.4354~
## # ... with 2,862 more rows, and 950 more variables: wtss <dbl+lbl>,
## #   wrkwayup <dbl+lbl>, wrkstat <dbl+lbl>, wrkslf <dbl+lbl>,
## #   wrkshift <dbl+lbl>, ...
```

```
# reorder the columns in increasing length of their names
gss %>% select(order(str_length(names(.))))
```

```
## # A tibble: 2,867 x 960
##      id age    god   jew   sex   tax   mar1  mar2  mar3  mar4  mar5  mar6
##   <dbl> <dbl+> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1 47     2     <NA>  1     2     1     1     5     5     <NA>  <NA>
## 2     2 61     <NA>  <NA>  1     <NA>  4     <NA>  <NA>  <NA>  <NA>  <NA>
## 3     3 72     6     <NA>  1     1     1     1     <NA>  <NA>  <NA>  <NA>
## 4     4 43     6     <NA>  2     1     1     1     5     4     5     5
## 5     5 55     1     <NA>  2     1     1     1     5     <NA>  <NA>  <NA>
## # ... with 2,862 more rows, and 948 more variables: mar7 <dbl+lbl>,
## #   mar8 <dbl+lbl>, mar9 <dbl+lbl>, age3 <dbl+lbl>, aged <dbl+lbl>, ...
```

```
# reorder the columns randomly
gss %>% select(sample(ncol(.)))
```

```
## # A tibble: 2,867 x 960
##   genegen  suineg2 scitext kidsinhh xmovie spother hlthstrt helpful spfund
##   <dbl+lb> <dbl+l> <dbl+l> <dbl+lb> <dbl+> <dbl+l> <dbl+lb> <dbl+l> <dbl+>
```

```
## 1 5        <NA>    2       1        <NA>   <NA>    <NA>     <NA>    3
## 2 <NA>     5       6       2        2      <NA>    <NA>     2       <NA>
## 3 <NA>     <NA>    <NA>    2        2      <NA>    <NA>     2       2
## 4 1        <NA>    6       2        <NA>   <NA>    <NA>     <NA>    2
## 5 <NA>     <NA>    <NA>    2        2      <NA>    1        3       3
## # ... with 2,862 more rows, and 951 more variables: partyid <dbl+lbl>,
## #   vote12 <dbl+lbl>, relsp11 <dbl+lbl>, rsecjob <dbl+lbl>,
## #   artatt1 <dbl+lbl>, ...
```

Selected columns can be renamed.

```
gss %>% select(ID = id, EDUCATION = educ)
```

```
## # A tibble: 2,867 x 2
##       ID EDUCATION
##    <dbl> <dbl+lbl>
## 1     1 16
## 2     2 12
## 3     3 16
## 4     4 12
## 5     5 18
## # ... with 2,862 more rows
```

#### 3.3.1.3 `rename()` columns

There's not much to `rename()`. We use it to change the names of the columns in a dataframe. Unlike `select()`, which can also change names, `rename()` keeps all columns. We write `new_column_name = old_column_name`.

```
hmd_counts %>% rename(Age = age, Exposure = nEx, Deaths = nDx)
```

```
## # A tibble: 1,304,694 x 7
##    country sex    period   Age    nx Deaths Exposure
##    <chr>   <chr>   <int> <int> <int>  <dbl>    <dbl>
## 1 AUS     Female   1921     0     1  3842.   64052.
## 2 AUS     Female   1921     1     1   719.   59619.
## 3 AUS     Female   1921     2     1   330.   57126.
## 4 AUS     Female   1921     3     1   166.   57484.
## 5 AUS     Female   1921     4     1   190.   58407.
## # ... with 1.305e+06 more rows
```

If we want to use spaces in the column names (not recomended though) we have to surround the new name with 'backticks'.

```
hmd_counts %>% rename(`Person-years exposure` = nEx, `Number of deaths` = nDx)
```

```
## # A tibble: 1,304,694 x 7
##    country sex    period   age    nx `Number of deaths` `Person-years expo~
##    <chr>   <chr>   <int> <int> <int>              <dbl>               <dbl>
## 1 AUS     Female   1921     0     1              3842.              64052.
## 2 AUS     Female   1921     1     1               719.              59619.
## 3 AUS     Female   1921     2     1               330.              57126.
## 4 AUS     Female   1921     3     1               166.              57484.
## 5 AUS     Female   1921     4     1               190.              58407.
## # ... with 1.305e+06 more rows
```

### 3.3.2 Row based transforms

#### 3.3.2.1 `filter()` rows

You can create subsets of the rows in your data using `filter()`.

```
# return rows of `euro_regio` where year is equal to 2016
euro_regio %>% filter(year == 2016)
```

```
## # A tibble: 342 x 21
##   country_code country_name nuts2_code nuts2_name     year     pop popdens
##   <chr>        <chr>        <chr>      <chr>         <int>   <dbl>   <dbl>
## 1 AL           Albania      AL01       Veri           2016  848059    NA
## 2 AL           Albania      AL02       Qender         2016 1110562    NA
## 3 AL           Albania      AL03       Jug            2016  927405    NA
## 4 AL           Albania      ALXX       ALXX Not reg~  2016      NA    NA
## 5 AT           Austria      AT11       Burgenland (~  2016  291011    77.1
## # ... with 337 more rows, and 14 more variables: births <dbl>,
## #   deaths <dbl>, natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>,
## #   ...
```

In `filter()` you specify logical tests which are evaluated for each row. If the condition evaluates to `TRUE` for a row, the row is returned, if it evaluates to `FALSE` the row is not returned. A row is also not returned if the condition evaluates to `NA`

Multiple filtering conditions are separated by a comma. The comma acts as a logical "AND", i.e. the `&` operator, which only returns `TRUE` if *all* the conditions are `TRUE`.

```
# return rows where year is equal to 2016 and region is Catalonia
euro_regio %>% filter(year == 2016, nuts2_code == 'ES51')
```

```
## # A tibble: 1 x 21
##   country_code country_name nuts2_code nuts2_name  year     pop popdens
##   <chr>        <chr>        <chr>      <chr>      <int>   <dbl>   <dbl>
## 1 ES           Spain        ES51       Cataluña    2016 7408290    232.
## # ... with 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

```
# this is the same as above
euro_regio %>% filter(year == 2016 & nuts2_code == 'ES51')
```

```
## # A tibble: 1 x 21
##   country_code country_name nuts2_code nuts2_name  year     pop popdens
##   <chr>        <chr>        <chr>      <chr>      <int>   <dbl>   <dbl>
## 1 ES           Spain        ES51       Cataluña    2016 7408290    232.
## # ... with 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

All of R's logical operators can be used in `filter()` as well. You use `<`, `>`, `>=`, `<=` for magnitude comparisons.

```
# is 1,2,3,4 larger than 1?
1:4 > 1
```

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

```
# return all rows where unemployment is higher than 30%
euro_regio %>% filter(unemp > 30)
```

```
## # A tibble: 27 x 21
```

```
##   country_code country_name nuts2_code nuts2_name      year    pop popdens
##   <chr>        <chr>        <chr>      <chr>          <int>  <dbl>   <dbl>
## 1 EL           Greece       EL52       Kentriki Make~  2013 1.91e6    99.6
## 2 EL           Greece       EL53       Dytiki Makedo~  2013 2.81e5    29.6
## 3 EL           Greece       EL53       Dytiki Makedo~  2015 2.76e5    29.1
## 4 EL           Greece       EL53       Dytiki Makedo~  2016 2.74e5    29.5
## 5 ES           Spain        ES43       Extremadura     2012 1.10e6    27.1
## # ... with 22 more rows, and 14 more variables: births <dbl>,
## #   deaths <dbl>, natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>,
## #   ...
```

```
# is 1,2,3,4 smaller than 1?
1:4 < 1
```

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

```
# return all rows where net-migration rate is lower than -20%
euro_regio %>% filter(netmigrate < -20)
```

```
## # A tibble: 13 x 21
##   country_code country_name nuts2_code nuts2_name      year    pop popdens
##   <chr>        <chr>        <chr>      <chr>          <int>  <dbl>   <dbl>
## 1 LT           Lithuania    LT00       Lietuva (NUTS~  2010 3.14e6    49.4
## 2 TR           Turkey       TR33       Manisa, Afyon~  2011 3.01e6    66.5
## 3 TR           Turkey       TR81       Zonguldak, Ka~  2011 1.04e6   108.
## 4 TR           Turkey       TRA1       Erzurum, Erzi~  2013 1.07e6    26.2
## 5 TR           Turkey       TRA2       Agri, Kars, I~  2009 1.14e6    37.9
## # ... with 8 more rows, and 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

```
# is 1,2,3,4 smaller or equal to 1?
1:4 <= 1
```

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

```
# return all rows where life-expectancy is 75 years or less
euro_regio %>% filter(lifeexp <= 75)
```

```
## # A tibble: 274 x 21
##   country_code country_name nuts2_code nuts2_name      year    pop popdens
##   <chr>        <chr>        <chr>      <chr>          <int>  <dbl>   <dbl>
## 1 BG           Bulgaria     BG31       Severozapaden   2005     NA    49.5
## 2 BG           Bulgaria     BG31       Severozapaden   2006 934755    48.5
## 3 BG           Bulgaria     BG31       Severozapaden   2007 916308    47.6
## 4 BG           Bulgaria     BG31       Severozapaden   2008 898371    46.7
## 5 BG           Bulgaria     BG31       Severozapaden   2009 881573    45.8
## # ... with 269 more rows, and 14 more variables: births <dbl>,
## #   deaths <dbl>, natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>,
## #   ...
```

```
# is 1,2,3,4 greater than or equal to 1?
1:4 >= 1
```

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

```
# return all rows where life-expectancy is 85 years or more
euro_regio %>% filter(lifeexp >= 85)
```

```
## # A tibble: 2 x 21
```

```
##   country_code country_name nuts2_code nuts2_name      year    pop popdens
##   <chr>        <chr>        <chr>      <chr>          <int>  <dbl>   <dbl>
## 1 CH           Switzerland  CH07       Ticino          2016 3.52e5    129.
## 2 ES           Spain        ES30       Comunidad de ~  2016 6.42e6    809.
## # ... with 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

Boolean logic is implemented via `&` (and), `|` (or), and `xor()` (exclusive or).

```
# is 1,2,3,4 greater than 1 AND is 1,2,3,4 smaller than 1?
(1:4 > 1) & (1:4 < 1)
```

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

```
# return all rows where unempolyment is higher than 15% and
# netmigration is lower than -15%
euro_regio %>% filter(unemp > 15, netmigrate < -15)
```

```
## # A tibble: 6 x 21
##   country_code country_name nuts2_code nuts2_name      year    pop popdens
##   <chr>        <chr>        <chr>      <chr>          <int>  <dbl>   <dbl>
## 1 CY           Cyprus       CY00       Kypros          2014 8.58e5    92.5
## 2 LT           Lithuania    LT00       Lietuva (NUTS~  2010 3.14e6    49.4
## 3 LV           Latvia       LV00       Latvija         2009 2.16e6    34.4
## 4 LV           Latvia       LV00       Latvija         2010 2.12e6    33.7
## 5 TR           Turkey       TRB2       Van, Mus, Bit~  2010 2.01e6    48.5
## # ... with 1 more row, and 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

```
# is 1,2,3,4 greater than 1 OR is 1,2,3,4 smaller than 1?
(1:4 > 1) | (1:4 < 1)
```

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

```
# return all rows where unempolyment is higher than 15% or
# netmigration is lower than -15%
euro_regio %>% filter(unemp > 15 | netmigrate < -15)
```

```
## # A tibble: 462 x 21
##   country_code country_name nuts2_code nuts2_name     year     pop popdens
##   <chr>        <chr>        <chr>      <chr>         <int>   <dbl>   <dbl>
## 1 AL           Albania      AL01       Veri           2014      NA     NA
## 2 AL           Albania      AL03       Jug            2014      NA     NA
## 3 BE           Belgium      BE10       Région de Br~  2006 1018804   6366.
## 4 BE           Belgium      BE10       Région de Br~  2007 1031215   6459.
## 5 BE           Belgium      BE10       Région de Br~  2008 1048491   6575.
## # ... with 457 more rows, and 14 more variables: births <dbl>,
## #   deaths <dbl>, natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>,
## #   ...
```

```
# is only one of those statements true?
# 1,2,3,4 greater than 1; 1,2,3,4 smaller than 5
xor(1:4 > 1, 1:4 < 5)
```

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

```
# return all rows where unempolyment is higher than 15% or
# netmigration is lower than -15% but not both
euro_regio %>% filter(xor(unemp > 15, netmigrate < -15))
```

```
## # A tibble: 404 x 21
##   country_code country_name nuts2_code nuts2_name      year    pop popdens
##   <chr>        <chr>        <chr>      <chr>          <int>  <dbl>   <dbl>
## 1 BE           Belgium      BE10       Région de Bru~  2006 1.02e6   6366.
## 2 BE           Belgium      BE10       Région de Bru~  2007 1.03e6   6459.
## 3 BE           Belgium      BE10       Région de Bru~  2008 1.05e6   6575.
## 4 BE           Belgium      BE10       Région de Bru~  2009 1.07e6   6702.
## 5 BE           Belgium      BE10       Région de Bru~  2010 1.09e6   6902
## # ... with 399 more rows, and 14 more variables: births <dbl>,
## #   deaths <dbl>, natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>,
## #   ...
```

The `%in%` operator lets you filter rows based on membership `%in%` a set.

```
# is 1,2,3,4 part of the set (2,4)
1:4 %in% c(2,4)
```

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

```
# return all rows for Germany, United Kingdom and France
euro_regio %>% filter(country_code %in% c('DE', 'UK', 'FR'))
```

```
## # A tibble: 1,404 x 21
##   country_code country_name    nuts2_code nuts2_name  year     pop popdens
##   <chr>        <chr>           <chr>      <chr>      <int>   <dbl>   <dbl>
## 1 DE           Germany (until~ DE11       Stuttgart   2005      NA    379.
## 2 DE           Germany (until~ DE11       Stuttgart   2006 4007373    380.
## 3 DE           Germany (until~ DE11       Stuttgart   2007 4005380    380.
## 4 DE           Germany (until~ DE11       Stuttgart   2008 4007095    380.
## 5 DE           Germany (until~ DE11       Stuttgart   2009 4006313    379.
## # ... with 1,399 more rows, and 14 more variables: births <dbl>,
## #   deaths <dbl>, natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>,
## #   ...
```

You can use any combination of functions within `filter()` as long as they return a logical vector as long as the input data frame. This makes it possible to…

```
# ... return the region with the highest population count in 2015
euro_regio %>%
  filter(year == 2015) %>%
  filter(pop == max(pop, na.rm = TRUE))
```

```
## # A tibble: 1 x 21
##   country_code country_name nuts2_code nuts2_name  year      pop popdens
##   <chr>        <chr>        <chr>      <chr>      <int>    <dbl>   <dbl>
## 1 TR           Turkey       TR10       Istanbul    2015 14377018   2794.
## # ... with 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

```
# ... return the regions with the lowest per head income in 2015, ranks 1 to 3
euro_regio %>%
  filter(year == 2015) %>%
  filter(min_rank(income) <= 3)
```

```
## # A tibble: 3 x 21
##   country_code country_name nuts2_code nuts2_name      year    pop popdens
##   <chr>        <chr>        <chr>      <chr>          <int>  <dbl>   <dbl>
## 1 BG           Bulgaria     BG31       Severozapaden   2015 7.97e5    42.5
```

```
## 2 BG           Bulgaria     BG42       Yuzhen tsentr~  2015 1.45e6    66
## 3 FR           France       FRA5       Mayotte (NUTS~  2015 2.30e5   628.
## # ... with 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

```
# ... return the regions with the highest per head income in 2015, ranks 1 to 3
euro_regio %>%
  filter(year == 2015) %>%
  filter(min_rank(desc(income)) <= 3)
```

```
## # A tibble: 3 x 21
##   country_code country_name     nuts2_code nuts2_name  year    pop popdens
##   <chr>        <chr>            <chr>      <chr>      <int>  <dbl>   <dbl>
## 1 DE           Germany (until ~ DE21       Oberbayern  2015 4.52e6    260.
## 2 LU           Luxembourg       LU00       Luxembourg  2015 5.63e5    220.
## 3 UK           United Kingdom   UKI3       Inner Lon~  2015 1.13e6  10469.
## # ... with 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

```
# ... return the regions with the highest absolute net-migration rate in 2015,
# ranks 1 to 3
euro_regio %>%
  filter(year == 2015) %>%
  filter(min_rank(desc(abs(netmigrate))) <= 3)
```

```
## # A tibble: 3 x 21
##   country_code country_name    nuts2_code nuts2_name   year    pop popdens
##   <chr>        <chr>           <chr>      <chr>       <int>  <dbl>   <dbl>
## 1 DE           Germany (until~ DEB2       Trier        2015 5.22e5   107.
## 2 MT           Malta           MT00       Malta        2015 4.40e5  1408.
## 3 TR           Turkey          TRA2       Agri, Kars~  2015 1.14e6    37.8
## # ... with 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

```
# ... return the >99 percentile regions by life expectancy in 2015
euro_regio %>%
  filter(year == 2015) %>%
  filter(cume_dist(lifeexp) >= 0.99)
```

```
## # A tibble: 4 x 21
##   country_code country_name nuts2_code nuts2_name      year    pop popdens
##   <chr>        <chr>        <chr>      <chr>          <int>  <dbl>   <dbl>
## 1 ES           Spain        ES22       Comunidad For~  2015 6.36e5    61.6
## 2 ES           Spain        ES30       Comunidad de ~  2015 6.39e6   804
## 3 FR           France       FR10       Île de France   2015 1.21e7  1008.
## 4 IT           Italy        ITH2       Provincia Aut~  2015 5.37e5    86.6
## # ... with 14 more variables: births <dbl>, deaths <dbl>,
## #   natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>, ...
```

Special care has to be taken when filtering `NA` values. One may be inclined to return all rows including `NA` in the variable `pop` by writing

```
euro_regio %>% filter(pop == NA)
```

```
## # A tibble: 0 x 21
## # ... with 21 variables: country_code <chr>, country_name <chr>,
## #   nuts2_code <chr>, nuts2_name <chr>, year <int>, ...
```

Zero rows are returned because for R `NA == NA` always returns `NA` instead of `TRUE` and `filter()` does not return rows for which a condition evaluates to `NA`. If we want to test for `NA` we must use the function `is.na()` which returns `TRUE` whenever an NA is encountered and `FALSE` otherwise.

```
euro_regio %>% filter(is.na(pop))
```

```
## # A tibble: 683 x 21
##   country_code country_name nuts2_code nuts2_name  year   pop popdens
##   <chr>        <chr>        <chr>      <chr>      <int> <dbl>   <dbl>
## 1 AL           Albania      AL01       Veri        2005    NA      NA
## 2 AL           Albania      AL01       Veri        2006    NA      NA
## 3 AL           Albania      AL01       Veri        2007    NA      NA
## 4 AL           Albania      AL01       Veri        2008    NA      NA
## 5 AL           Albania      AL01       Veri        2009    NA      NA
## # ... with 678 more rows, and 14 more variables: births <dbl>,
## #   deaths <dbl>, natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>,
## #   ...
```

#### 3.3.2.2 `arrange()` rows

`arrange()` re-orders the rows of a dataframe according to the values of one or more variables within that dataframe.

```
# order data frame by increasing date
eu_timeuse_tot %>% arrange(year)
```

```
## # A tibble: 1,671 x 8
##   country_code country_name  year activity_code activity_name   prtcp_rate
##   <chr>        <chr>        <int> <chr>         <chr>                <dbl>
## 1 BE           Belgium       2000 AC0           Personal care        100
## 2 BE           Belgium       2000 AC01          Sleep                100
## 3 BE           Belgium       2000 AC02          Eating                99.5
## 4 BE           Belgium       2000 AC03          Other and/or u~       97.6
## 5 BE           Belgium       2000 AC1_TR        Employment, re~       34.5
## # ... with 1,666 more rows, and 2 more variables: prtcp_time_min <dbl>,
## #   time_spent_min <dbl>
```

If multiple variables are specified, the dataframe is re-ordered in the sequence of specification. This behavior makes it possible to design useful tables, i.e. our `eu_timeuse` data contains the time spent each day on different activities by type of activity, country, and year. Sorting by `activity`, `country` and (at last position) `year` gives a table that allows for quick comparisions along the time axis. Having `country` last would allow for quick comparisions among coutries and so on.

```
# order data frame by activity, country and year
eu_timeuse_tot %>% arrange(activity_name, country_name, year)
```

```
## # A tibble: 1,671 x 8
##   country_code country_name  year activity_code activity_name   prtcp_rate
##   <chr>        <chr>        <int> <chr>         <chr>                <dbl>
## 1 AT           Austria       2010 AC1B          Activities rel~       22.6
## 2 BE           Belgium       2000 AC1B          Activities rel~        1.4
## 3 BE           Belgium       2010 AC1B          Activities rel~       12.5
## 4 BG           Bulgaria      2000 AC1B          Activities rel~        7.1
## 5 EE           Estonia       2000 AC1B          Activities rel~       10.9
## # ... with 1,666 more rows, and 2 more variables: prtcp_time_min <dbl>,
## #   time_spent_min <dbl>
```

```
# order data frame by activity, year, and country
eu_timeuse_tot %>% arrange(activity_name, year, country_name)
```

```
## # A tibble: 1,671 x 8
##   country_code country_name  year activity_code activity_name   prtcp_rate
##   <chr>        <chr>        <int> <chr>         <chr>                <dbl>
## 1 BE           Belgium       2000 AC1B          Activities rel~        1.4
## 2 BG           Bulgaria      2000 AC1B          Activities rel~        7.1
## 3 EE           Estonia       2000 AC1B          Activities rel~       10.9
## 4 FI           Finland       2000 AC1B          Activities rel~        7.2
## 5 FR           France        2000 AC1B          Activities rel~        1.6
## # ... with 1,666 more rows, and 2 more variables: prtcp_time_min <dbl>,
## #   time_spent_min <dbl>
```

By default character variables are ordered in increasing alphabetical order and numeric variables in increasing numerical order. The `desc()` function allows to reverse that behavior.

```
# order data frame by reverse alphabetical activity, and alphabetical country
eu_timeuse_tot %>% arrange(desc(activity_name), country_name)
```

```
## # A tibble: 1,671 x 8
##   country_code country_name  year activity_code activity_name   prtcp_rate
##   <chr>        <chr>        <int> <chr>         <chr>                <dbl>
## 1 AT           Austria       2010 AC344         Walking the dog        6.5
## 2 BE           Belgium       2000 AC344         Walking the dog        7.1
## 3 BE           Belgium       2010 AC344         Walking the dog        8.1
## 4 BG           Bulgaria      2000 AC344         Walking the dog        2.2
## 5 EE           Estonia       2000 AC344         Walking the dog        8.5
## # ... with 1,666 more rows, and 2 more variables: prtcp_time_min <dbl>,
## #   time_spent_min <dbl>
```

#### 3.3.2.3 `slice()` rows

Rows are selected with `slice()`. The rows in a dataframe are indexed 1 to n, with n being the total number of rows. The `slice()` function takes a single integer or a vector of integers and returns the rows with the correponding index.

```
# extract the first row
hmd_counts %>% slice(1)
```

```
## # A tibble: 1 x 7
##   country sex    period   age    nx   nDx    nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     Female   1921     0     1 3842. 64052.
```

```
# extract the last row
hmd_counts %>% slice(n())
```

```
## # A tibble: 1 x 7
##   country sex   period   age    nx   nDx   nEx
##   <chr>   <chr>  <int> <int> <int> <dbl> <dbl>
## 1 USA     Total   2016   110    NA    94  165.
```

```
# extract rows 20 to 40
hmd_counts %>% slice(20:40)
```

```
## # A tibble: 21 x 7
```

```
##   country sex    period   age    nx   nDx    nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     Female   1921    19     1  101. 45909.
## 2 AUS     Female   1921    20     1  138. 46387.
## 3 AUS     Female   1921    21     1  118. 46437.
## 4 AUS     Female   1921    22     1  116. 45769.
## 5 AUS     Female   1921    23     1  149. 45746.
## # ... with 16 more rows
```

Functions which return integers can be used inside `slice()`.

```
# extract every 10th row
hmd_counts%>% slice(seq(1, n(), 10))
```

```
## # A tibble: 130,470 x 7
##   country sex    period   age    nx    nDx    nEx
##   <chr>   <chr>   <int> <int> <int>  <dbl>  <dbl>
## 1 AUS     Female   1921     0     1 3842.  64052.
## 2 AUS     Female   1921    10     1   69.0 55240.
## 3 AUS     Female   1921    20     1  138.  46387.
## 4 AUS     Female   1921    30     1  183.  46315.
## 5 AUS     Female   1921    40     1  198.  35317.
## # ... with 1.305e+05 more rows
```

```
# extract 50 random rows
hmd_counts%>% slice(sample(1:n(), 50))
```

```
## # A tibble: 50 x 7
##   country sex   period   age    nx   nDx    nEx
##   <chr>   <chr>  <int> <int> <int> <dbl>  <dbl>
## 1 PRT     Male    1965   109     1    0      0
## 2 DEUTW   Total   1963    93     1 1973.  5101.
## 3 BGR     Total   2006    92     1  773   3069.
## 4 ITA     Total   1964    87     1 8119  39648.
## 5 GBR_NIR Male    1987    66     1  187   6266.
## # ... with 45 more rows
```

```
# randomly shuffle all rows
hmd_counts %>% slice(sample(1:n()))
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx   nDx     nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>   <dbl>
## 1 FRATNP  Male     1879    56     1 4366. 184856.
## 2 FRATNP  Total    1898    13     1 2136. 674579.
## 3 ISL     Female   2002     0     1    2    2015.
## 4 BEL     Total    1863    63     1 1192.  34703.
## 5 JPN     Female   2015    45     1  851. 905197.
## # ... with 1.305e+06 more rows
```

### 3.3.3 Excercise: Basic verbs

- Fix the following code:

```
hmd_counts %>%
  filter(country = 'AUS', age = '0', sex = 'Female') %>%
```

```
  mutate(hmd_counts, period_width = diff(period))
#hmd_counts %>%
#  filter(country == 'AUS', age == 0, sex == 'Female') %>%
#  mutate(period_width = c(diff(period), NA))

gss %>% filter(bigbang == NA)
#gss %>% filter(is.na(NA))
```

- Filter `hmd_counts` such that it contains all Swedish life-tables and only life-tables for years 2000+ for countries other than Sweden.
- Why do the results differ?

```
euro_regio %>% filter(year == 2015, rank(-income) <= 3)
euro_regio %>%
  filter(year == 2015) %>%
  filter(rank(-income) <= 3)
```

- With `gss` select
- the first 10 columns
- the last 10 columns
- every 5th column from 1 to 100
- every column where the name does not contain a number
- all columns where the names contain a phrase of your choice

## 3.4 Data pipelines

We can chain multiple function and transformations steps into a *data analysis pipeline*. This is a great approach for clear, fast, interactive data analysis.

This is what we need to know in order to build pipelines:

- By default, the object on the left of the pipe operator (`%>%`) is passed onto the first argument of the function on the right.

```
# x is the first argument of the mean function...
mean(x = 1:10)
```

```
## [1] 5.5
```

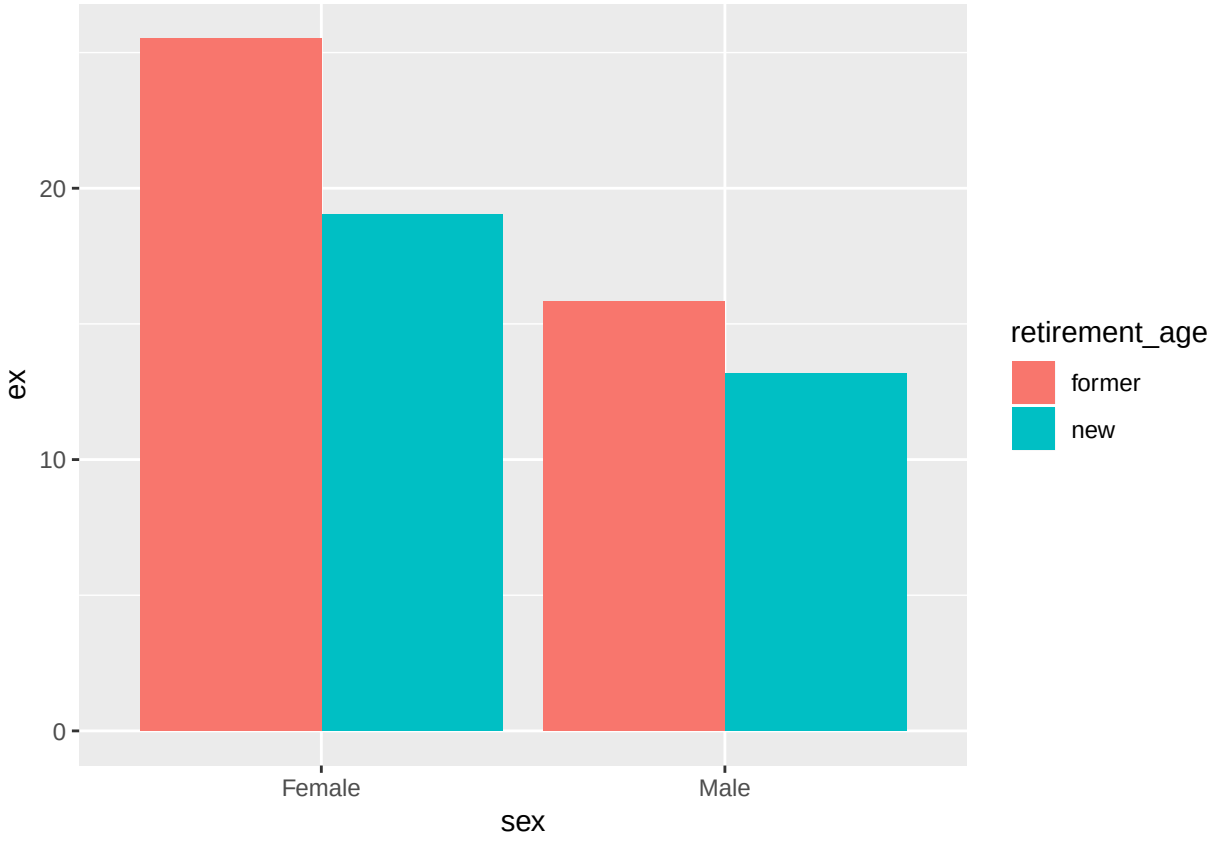
```
# ... 1:10 gets passed to x
1:10 %>% mean()
```

```
## [1] 5.5
```

Here's a pipeline which begins with raw data and ends with a plot after some data transformations steps in between.

```
# remaining life-expectancy at former
# and new retirement age in Russia by sex
hmd %>%
  filter(period == 2014, country == 'RUS', sex != 'Total',
         sex == 'Male' & age %in% c(60, 65) |
           sex == 'Female' & age %in% c(55, 63)) %>%
  mutate(retirement_age = case_when(age %in% c(55, 60) ~ 'former',
                                    age %in% c(63, 65)~ 'new')) %>%
  select(sex, retirement_age, ex) %>%
```

```
  ggplot(aes(x = sex, y = ex, fill = retirement_age)) +
  geom_col(position = 'dodge')
```




- If we want to use the object on the left in other places than the first argument we can explicitly refer to it by using a dot (`.`). In that case the object on the left is only passed to the dot and not to the first argument.

```
# linear trend in life-expectancy at birth in Russia by sex
hmd %>%
  filter(country == 'RUS', sex != 'Total', age == 0) %>%
  mutate(period_std = period - min(period)) %>%
  glm(ex ~ period_std*sex, data = .)
```

```
##
## Call:  glm(formula = ex ~ period_std * sex, data = .)
##
## Coefficients:
##        (Intercept)          period_std             sexMale
##           72.72682             0.02300            -9.17573
## period_std:sexMale
##           -0.07103
##
## Degrees of Freedom: 111 Total (i.e. Null);  108 Residual
## Null Deviance:       3775
## Residual Deviance: 265.9     AIC: 424.7
```

- Surrounding an expression with curly braces `{}` surpresses the left-hand side input. Instead you must use the dot notation to refer to that input. The advantage is that the dot does not need to stand on

its own. It can be indexed like a regular R object.

```
# correlation between infant mortality and life-expectancy
hmd %>%
  filter(country == 'RUS', sex != 'Total', age == 0) %>%
  mutate(period_std = period - min(period)) %>%
  {cor(x = .$nmx, y = .$ex)}
```

```
## [1] -0.3190741
```

### 3.4.1 Excercise: Data pipelines

Rewrite the following expressions as pipes:

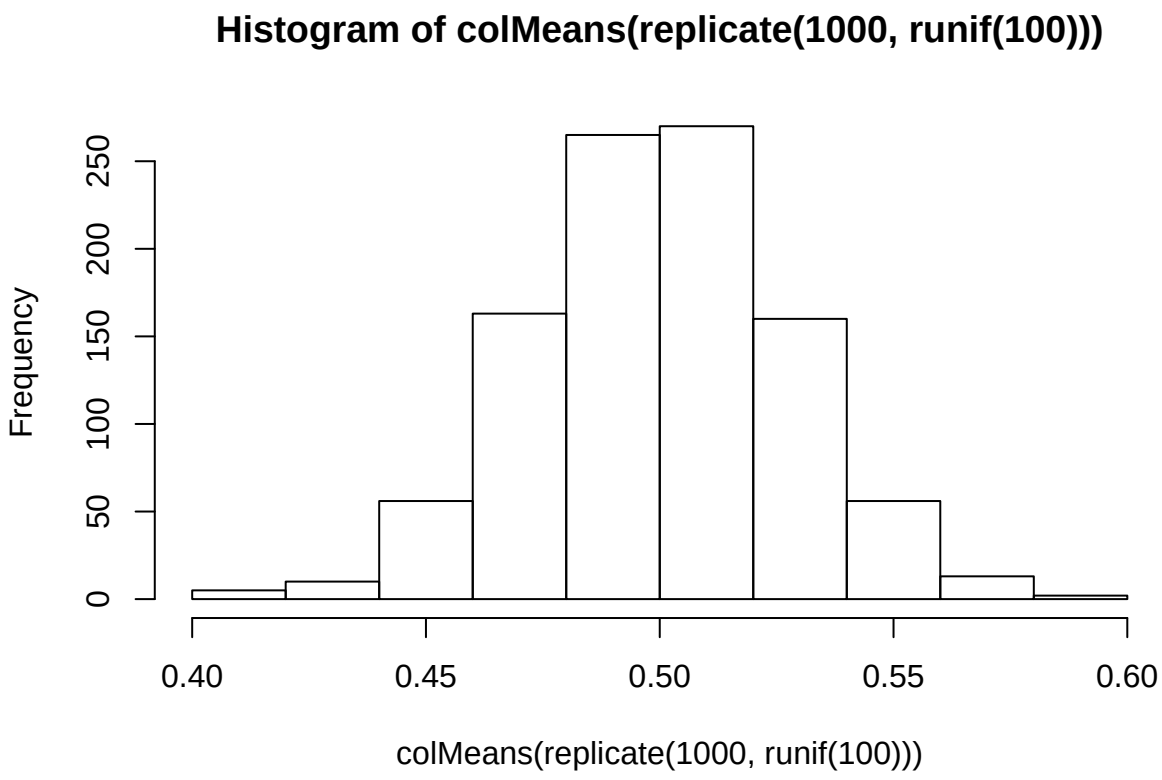
```
sum(1:100)
```

```
FALSE [1] 5050
```

```
#1:100 %>% sum()

sum(cumsum(1:100))
```

```
FALSE [1] 171700
```

```
#1:100 %>% cumsum() %>% sum()

hist(colMeans(replicate(1000, runif(100))))
```




```
#replicate(1000, runif(100)) %>% colMeans() %>% hist()

any(is.na(mtcars$cyl))
```

```
FALSE [1] FALSE
```

```
#mtcars$cyl %>% is.na() %>% any()

cor.test(~hp+mpg, data = mtcars)
```

```
FALSE
FALSE   Pearson's product-moment correlation
FALSE
FALSE data:  hp and mpg
FALSE t = -6.7424, df = 30, p-value = 1.788e-07
FALSE alternative hypothesis: true correlation is not equal to 0
FALSE 95 percent confidence interval:
FALSE  -0.8852686 -0.5860994
FALSE sample estimates:
FALSE        cor
FALSE -0.7761684
```

```
#mtcars %>% cor.test(~ hp+mpg, data = .)

cor.test(~hp+mpg, data = mtcars[1:10])
```

```
FALSE
FALSE   Pearson's product-moment correlation
FALSE
FALSE data:  hp and mpg
FALSE t = -6.7424, df = 30, p-value = 1.788e-07
FALSE alternative hypothesis: true correlation is not equal to 0
FALSE 95 percent confidence interval:
FALSE  -0.8852686 -0.5860994
FALSE sample estimates:
FALSE        cor
FALSE -0.7761684
```

```
#mtcars %>% {cor.test(~ hp+mpg, data = .[1:10])}

cor(x = mtcars$mpg, y = mtcars$hp)
```

```
FALSE [1] -0.7761684
```

```
#mtcars %>% {cor(x = .$mpg, y = .$hp)}

xtabs(~ gear + cyl, data = mtcars, subset = mtcars$gear > 4)
```

```
FALSE     cyl
FALSE gear 4 6 8
FALSE    5 2 1 2
```

```
#mtcars %>% xtabs(~gear + cyl, data = ., subset = .$gear > 4)
```

## 3.5 Tidy data

### 3.5.1 What is tidy data?

Many programming tasks become easier once the data is in a tidy format. But what is tidy data? Our working definition: **data needs to be a data frame** and **every variable of interest needs to be a separate column**. Let's explore what that means.

```
head(WorldPhones)
```

```
##      N.Amer Europe Asia S.Amer Oceania Africa Mid.Amer
## 1951  45939  21574 2876   1815    1646     89      555
## 1956  60423  29990 4708   2568    2366   1411      733
## 1957  64721  32510 5230   2695    2526   1546      773
## 1958  68484  35218 6662   2845    2691   1663      836
## 1959  71799  37598 6856   3000    2868   1769      911
## 1960  76036  40341 8220   3145    3054   1905     1008
```

Here's the number of telephone connections over time by continent. The data is not *tidy* because its not a *data frame*, it's a matrix with row and column names. This gives us headaches if we want to use ggplot to plot the data.

```
ggplot(WorldPhones)
```

```
## Error: `data` must be a data frame, or other object coercible by `fortify()`, not a numeric vector
```

We can easily fix this problem by converting the matrix to a data frame.

```
phones <- as.data.frame(WorldPhones)
```

Say we we want to plot the number of telephone connections over time by continent. This implies the following *variables of interest*:

```
* the number of telephone connections `n`
* the continent `cont`
* the year `year`
```

Problem is, *none* of these variables are explicitly given in our data frame. Of course the data is all there, just not in a format we can use (with ggplot). So the question is how to reshape the data into a form where all the variables of interest are separate columns in the data frame.

The easiest variable to make explicit is the year. It is given as rownames of the data frame. We take the rownames, convert them from character to integer type, and add them as the variable `year` to the data frame. We use the `tidyverse` function `mutate()` to add a new variable to a data frame.

```
phones <- mutate(phones, year = as.integer(rownames(phones)))
phones
```

```
##   N.Amer Europe Asia S.Amer Oceania Africa Mid.Amer year
## 1  45939  21574 2876   1815    1646     89      555 1951
## 2  60423  29990 4708   2568    2366   1411      733 1956
## 3  64721  32510 5230   2695    2526   1546      773 1957
## 4  68484  35218 6662   2845    2691   1663      836 1958
## 5  71799  37598 6856   3000    2868   1769      911 1959
## 6  76036  40341 8220   3145    3054   1905     1008 1960
## 7  79831  43173 9053   3338    3224   2005     1076 1961
```

That leaves us with the variables *"number of telephone connections"* and *"continent"* to make explicit. They shall become separate columns in the data frame. With the help of `gather()` we **transform from wide**

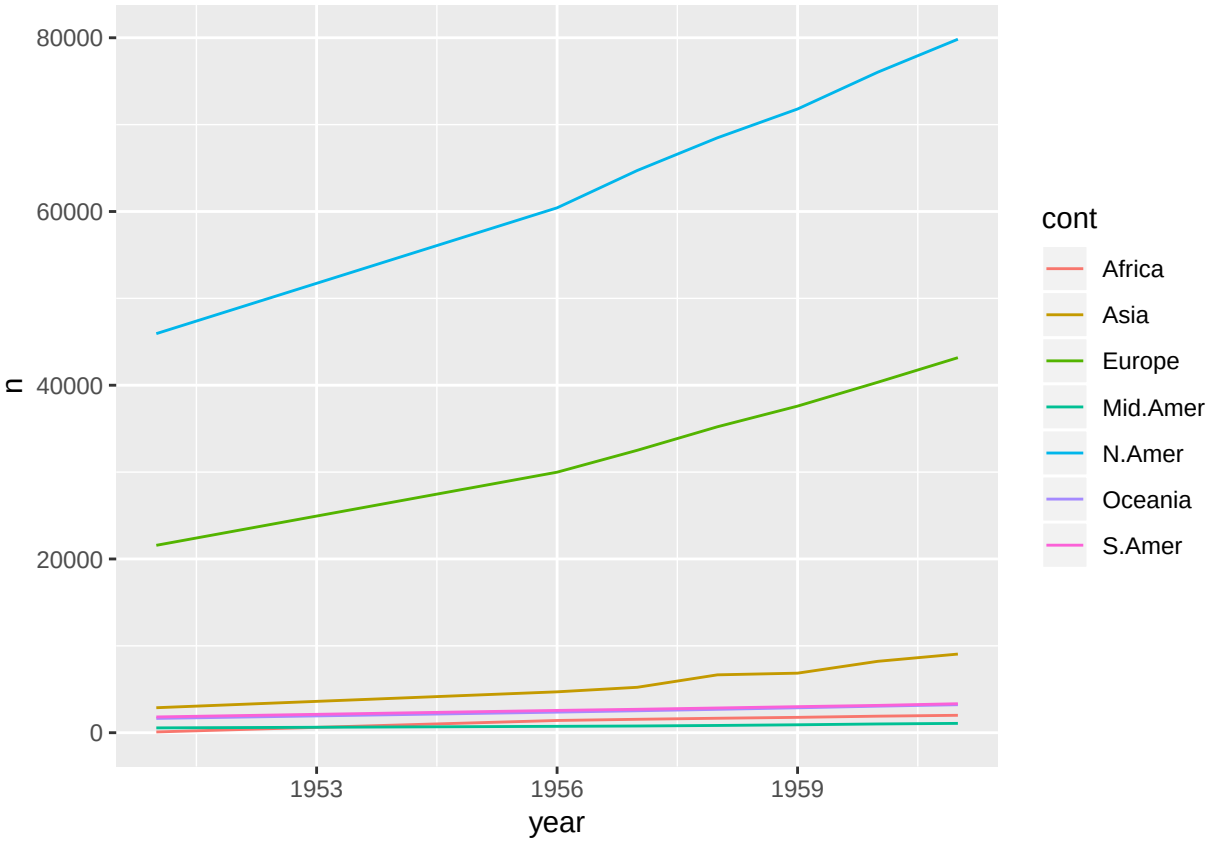
**to long format**.

```
phones <- gather(phones, key = cont, value = n, -year)
phones
```

```
##    year     cont     n
## 1  1951   N.Amer 45939
## 2  1956   N.Amer 60423
## 3  1957   N.Amer 64721
## 4  1958   N.Amer 68484
## 5  1959   N.Amer 71799
## 6  1960   N.Amer 76036
## 7  1961   N.Amer 79831
## 8  1951   Europe 21574
## 9  1956   Europe 29990
## 10 1957   Europe 32510
## 11 1958   Europe 35218
## 12 1959   Europe 37598
## 13 1960   Europe 40341
## 14 1961   Europe 43173
## 15 1951     Asia  2876
## 16 1956     Asia  4708
## 17 1957     Asia  5230
## 18 1958     Asia  6662
## 19 1959     Asia  6856
## 20 1960     Asia  8220
## 21 1961     Asia  9053
## 22 1951   S.Amer  1815
## 23 1956   S.Amer  2568
## 24 1957   S.Amer  2695
## 25 1958   S.Amer  2845
## 26 1959   S.Amer  3000
## 27 1960   S.Amer  3145
## 28 1961   S.Amer  3338
## 29 1951  Oceania  1646
## 30 1956  Oceania  2366
## 31 1957  Oceania  2526
## 32 1958  Oceania  2691
## 33 1959  Oceania  2868
## 34 1960  Oceania  3054
## 35 1961  Oceania  3224
## 36 1951   Africa    89
## 37 1956   Africa  1411
## 38 1957   Africa  1546
## 39 1958   Africa  1663
## 40 1959   Africa  1769
## 41 1960   Africa  1905
## 42 1961   Africa  2005
## 43 1951 Mid.Amer   555
## 44 1956 Mid.Amer   733
## 45 1957 Mid.Amer   773
## 46 1958 Mid.Amer   836
## 47 1959 Mid.Amer   911
## 48 1960 Mid.Amer  1008
## 49 1961 Mid.Amer  1076
```

We told the computer to look at all columns apart from `year` and transform them into the columns `cont` and `n`. `cont` holds the continent names for the variable `n`, the number of telephone connections. The continent names are taken from the original column names we *gathered* over. We now can plot our data easily.

```
ggplot(phones) +
  geom_line(aes(x = year, y = n, colour = cont))
```




### 3.5.2 Convert to data frame

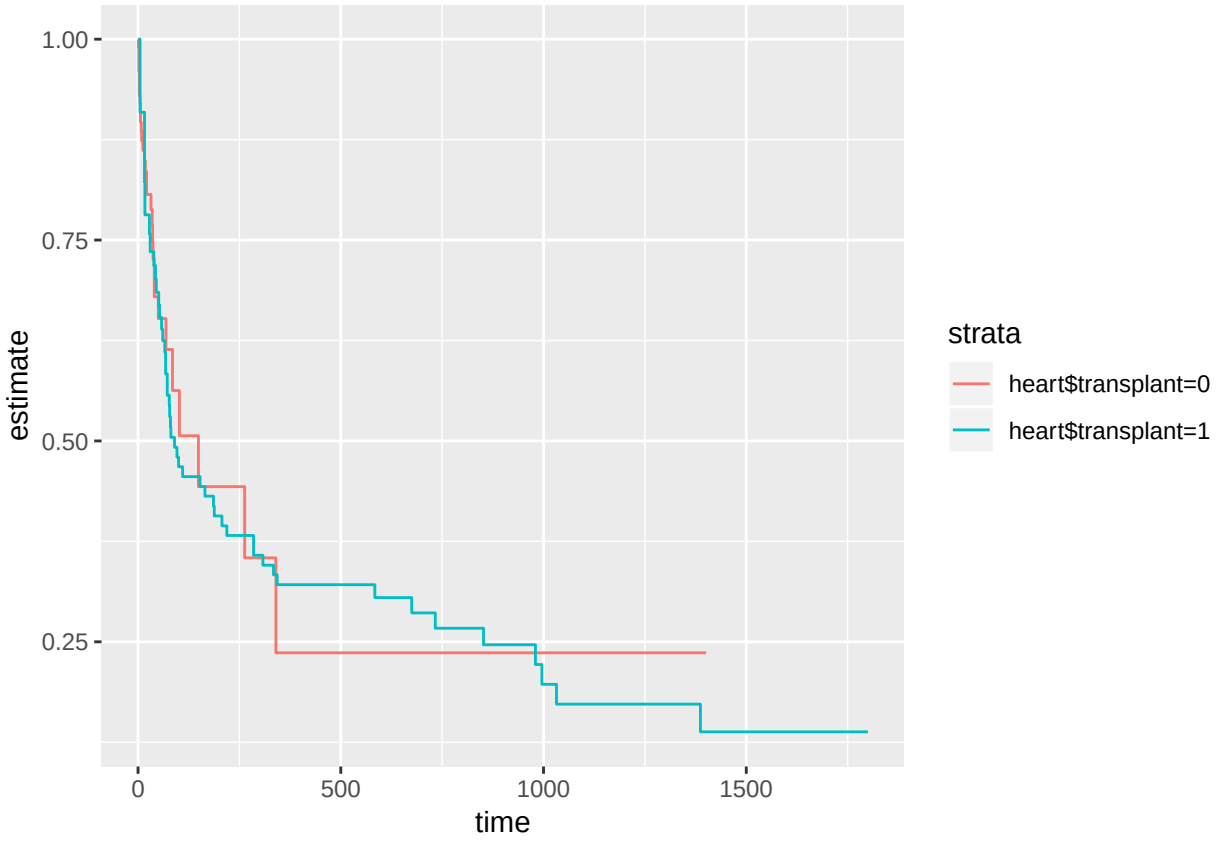
We "tidy" the output of the `survfit` function via the "broom" package.

```
library(survival)
surv <- survfit(
  Surv(time = heart$start,
       time2 = heart$stop,
       event = heart$event) ~ heart$transplant
)
surv
```

```
## Call: survfit(formula = Surv(time = heart$start, time2 = heart$stop,
##     event = heart$event) ~ heart$transplant)
##
##                      records n.max n.start events median 0.95LCL 0.95UCL
## heart$transplant=0       103   103       0     30    149      69      NA
## heart$transplant=1        69    45       0     45     90      53     334
```

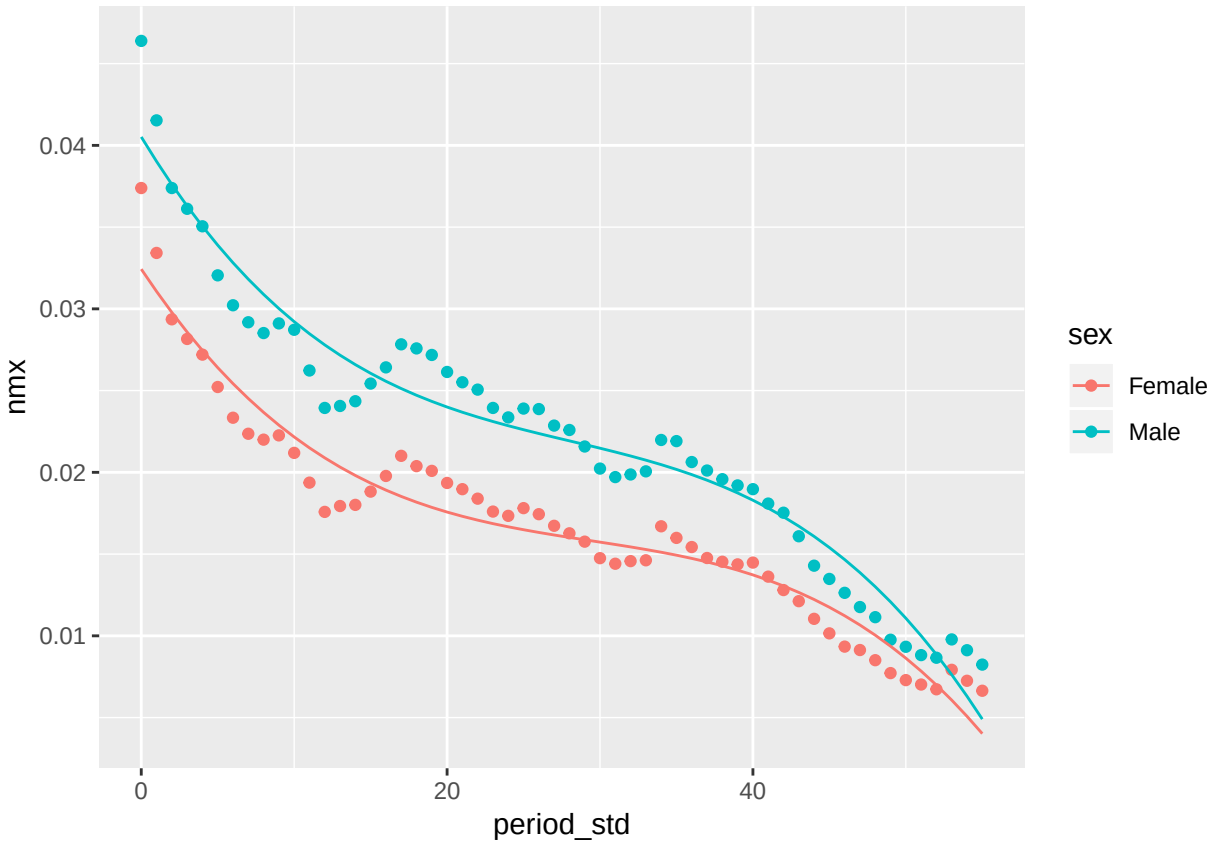
```
broom::tidy(surv) %>%
  ggplot(aes(x = time, y = estimate)) +
```

```
  geom_step(aes(colour = strata))
```




Dobson (1990) Page 93: Randomized Controlled Trial.

```
library(splines)

# linear trend in life-expectancy at birth in Russia by sex
hmd %>%
  filter(country == 'RUS', sex != 'Total', age == 0) %>%
  mutate(period_std = period - min(period)) %>%
  glm(nmx ~ period_std + bs(period_std, df = 3)*sex, data = .) %>%
  broom::augment() %>%
  ggplot(aes(x = period_std, color = sex)) +
  geom_point(aes(y = nmx)) +
  geom_line(aes(y = .fitted))
```




### 3.5.3 Long versus wide format

Each table has a *wide format* and a long format representation. The information content is the same in both formats. It's the layout that differs.

Here's a wide format table containing the explicit variables `Female` and `Male`.

```
wide <- data_frame(group = c("a", "b"), Female = 1:2, Male = 3:4)
```

The same table in long format representation containing the explicit variables `Sex` and `N`.

```
long <- gather(wide, key = Sex, value = N, -group)
long
```

```
## # A tibble: 4 x 3
##   group Sex        N
##   <chr> <chr>  <int>
## 1 a     Female     1
## 2 b     Female     2
## 3 a     Male       3
## 4 b     Male       4
```

If we want to go back to a wide format we can achieve that by using the function `spread()`.

```
spread(long, key = Sex, value = N)
```

```
## # A tibble: 2 x 3
##   group Female  Male
##   <chr>  <int> <int>
```

```
## 1 a          1     3
## 2 b          2     4
```

```
# common problems: no matching row was found
hmd %>%
  filter(period > 2005, country == 'RUS', age == 0) %>%
  spread(sex, ex) %>%
  select(country, period, age, Female, Male)
```

```
## # A tibble: 27 x 5
##   country period   age Female  Male
##   <chr>    <int> <int>  <dbl> <dbl>
## 1 RUS       2006     0   73.3    NA
## 2 RUS       2007     0   74.0    NA
## 3 RUS       2008     0   74.2    NA
## 4 RUS       2009     0   74.8    NA
## 5 RUS       2010     0   74.9    NA
## # ... with 22 more rows
```

```
# solution
hmd %>%
  filter(period > 2005, country == 'RUS', age == 0) %>%
  select(-(nDx:Tx)) %>%
  spread(sex, ex) %>%
  select(country, period, age, Female, Male)
```

```
## # A tibble: 9 x 5
##   country period   age Female  Male
##   <chr>    <int> <int>  <dbl> <dbl>
## 1 RUS       2006     0   73.3  60.4
## 2 RUS       2007     0   74.0  61.4
## 3 RUS       2008     0   74.2  61.9
## 4 RUS       2009     0   74.8  62.8
## 5 RUS       2010     0   74.9  63.1
## # ... with 4 more rows
```

### 3.5.4 Handling missing values

```
# turn implicit NAs into explicit NAs
eu_timeuse_tot %>% skimr::n_missing()
```

```
## [1] 17
```

```
eu_timeuse_tot %>% complete(activity_code, country_code, year) %>% skimr::n_missing()
```

```
## [1] 2862
```

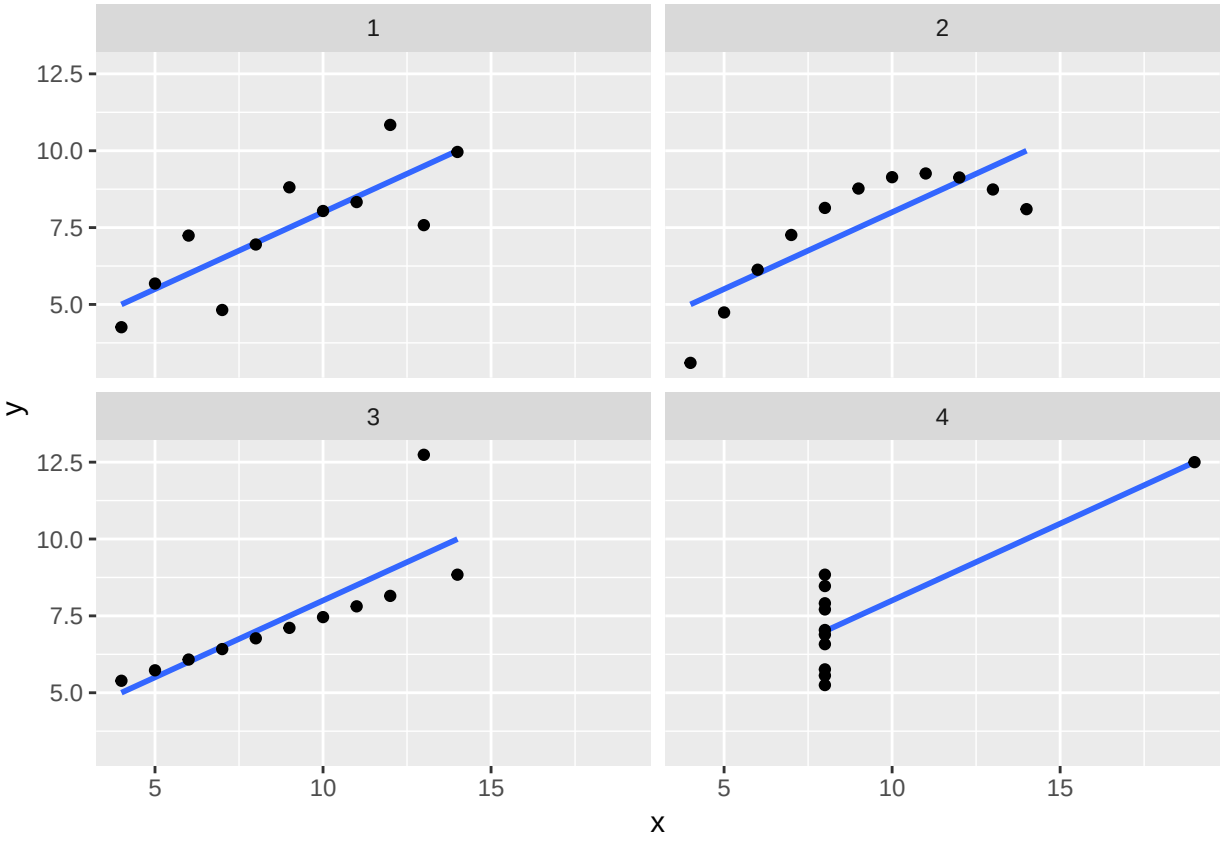
### 3.5.5 Recoding variables

### 3.5.6 Case studies in data cleaning

#### 3.5.6.1 Tidying Anscombe's quartet

Can you figure out what happens here? Try running the code yourself line by line.

```
anscombe %>%
  mutate(id = seq_along(x1)) %>%
  gather(... = -id) %>%
  separate(key, sep = 1, into = c("axis", "panel")) %>%
  spread(key = axis, value = value) %>%
  ggplot(aes(x = x, y = y)) +
  geom_smooth(method = "lm", se = FALSE) +
  geom_point() +
  facet_wrap(~panel)
```




#### 3.5.6.2 Tidying data on test-retest reliability

```
wide <- read_csv("https://raw.githubusercontent.com/jschoeley/idem_viz/master/ggplot_practical/03-tidy_
wide
```

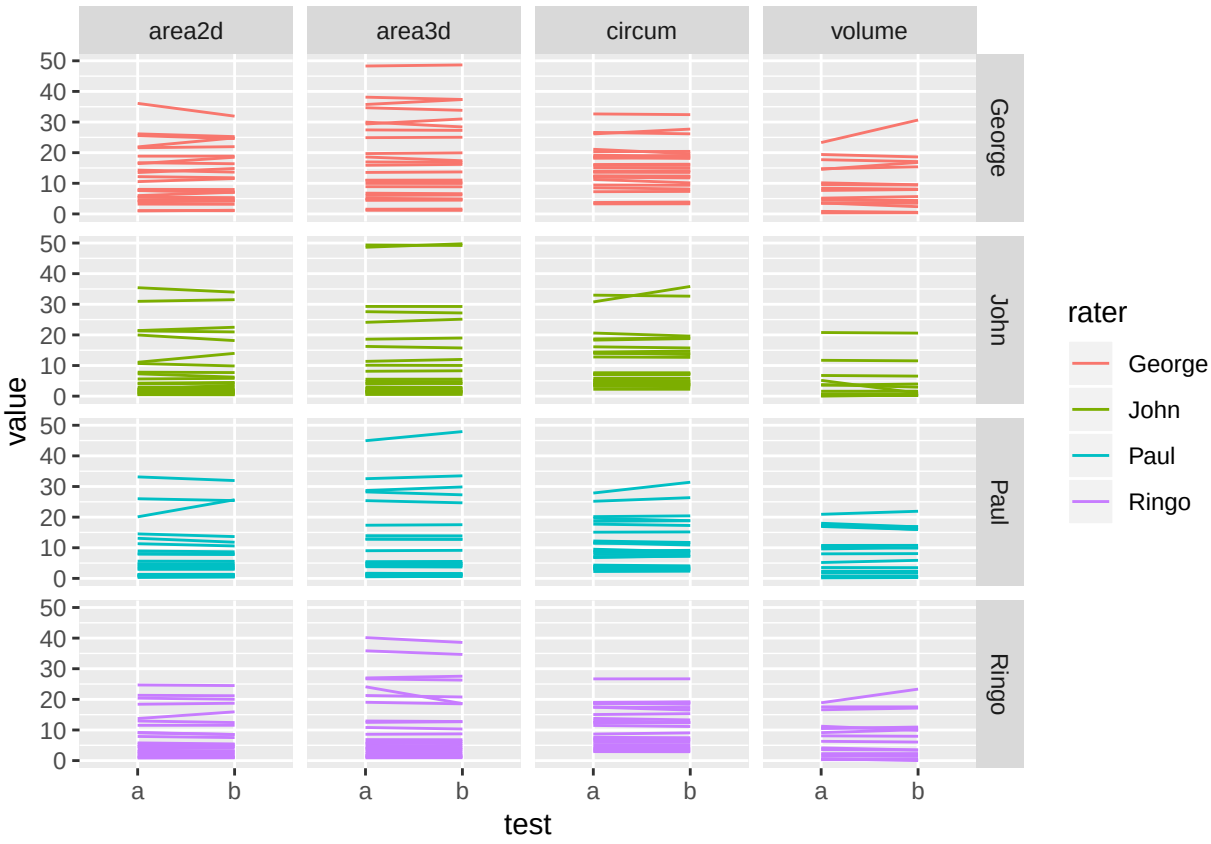
```
## # A tibble: 48 x 20
##   name_rater1 area2d_camera1a area2d_camera1b area3d_camera1a
##   <chr>                 <dbl>           <dbl>           <dbl>
## 1 John                  2.03            2.08            2.51
## 2 Paul                  0.300           0.420           0.510
## 3 John                  4.24            4.46            5.53
## 4 John                  2.87            3.08            4.03
## 5 John                  5.60            5.83            8.13
## # ... with 43 more rows, and 16 more variables: area3d_camera1b <dbl>,
## #   volume_camera1a <dbl>, volume_camera1b <dbl>, volume_gel1 <dbl>,
## #   circum_camera1a <dbl>, ...
```

```
long <-
  wide %>%
  # add a unique identifier to each row (each patient)
  mutate(id = 1:nrow(.)) %>%
  gather(key = type, value = value, -id, -name_rater1, -name_rater2) %>%
  separate(col = type, into = c("measurement", "method"), sep = "_") %>%
  mutate(rater = ifelse(grepl('1', method), name_rater1, name_rater2)) %>%
  separate(col = method, into = c("method", "test"), sep = "\\d") %>%
  mutate(test = ifelse(test == "", "a", test)) %>%
  # beautification
  select(id, rater, test, measurement, method, value) %>%
  arrange(id, measurement, rater, test)
long
```

```
## # A tibble: 864 x 6
##      id rater test  measurement method value
##   <int> <chr> <chr> <chr>       <chr>  <dbl>
## 1     1 John  a     area2d      camera  2.03
## 2     1 John  b     area2d      camera  2.08
## 3     1 Paul  a     area2d      camera  1.95
## 4     1 Paul  b     area2d      camera NA
## 5     1 John  a     area3d      camera  2.51
## # ... with 859 more rows
```

```
long %>%
  filter(method == "camera") %>%
  ggplot(aes(x = test, y = value)) +
  geom_line(aes(color= rater, group = id)) +
  facet_grid(rater~measurement)
```

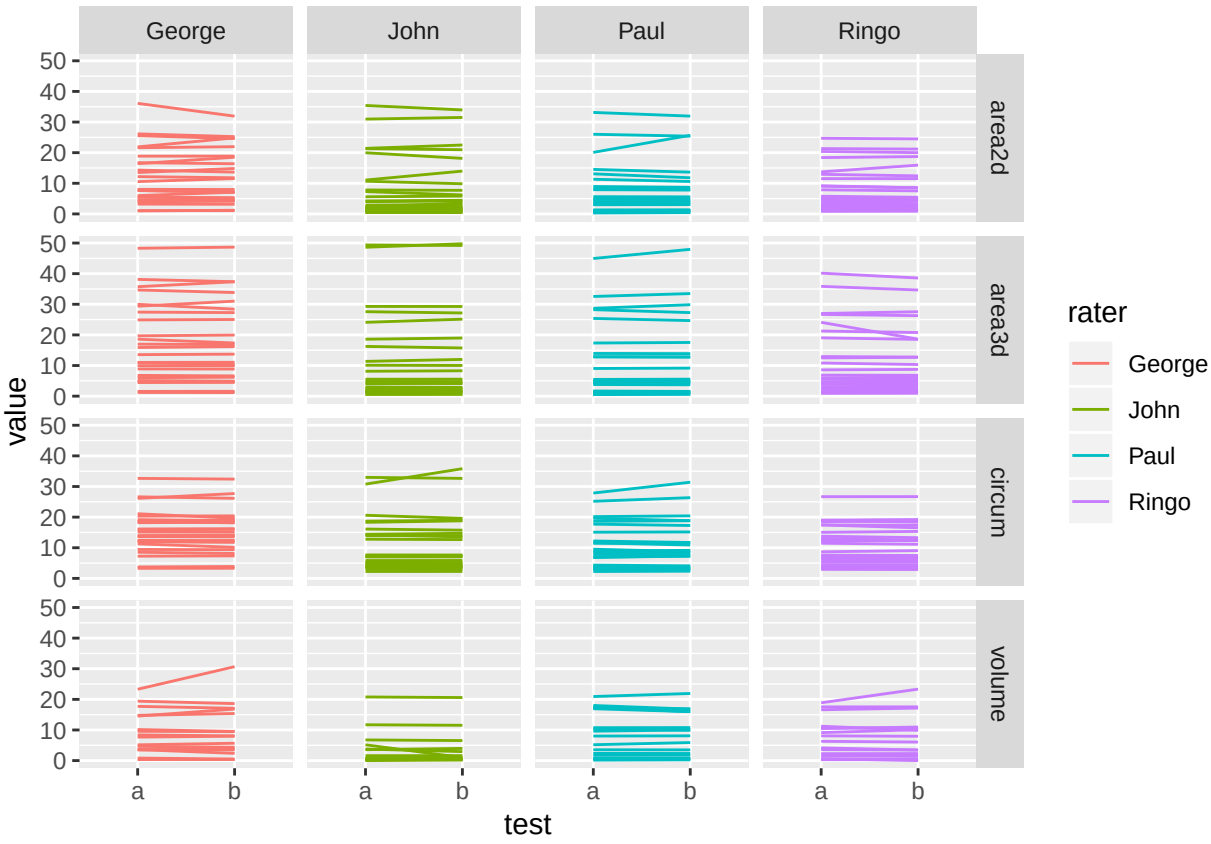
```
## Warning: Removed 30 rows containing missing values (geom_path).
```




Comparisions along the y-axis is easiest if the scales are aligned therefore it is easier to compare along the horizontal.

```
long %>%
  filter(method == "camera") %>%
  ggplot(aes(x = test, y = value)) +
  geom_line(aes(color= rater, group = id)) +
  facet_grid(measurement~rater)
```

```
## Warning: Removed 54 rows containing missing values (geom_path).
```

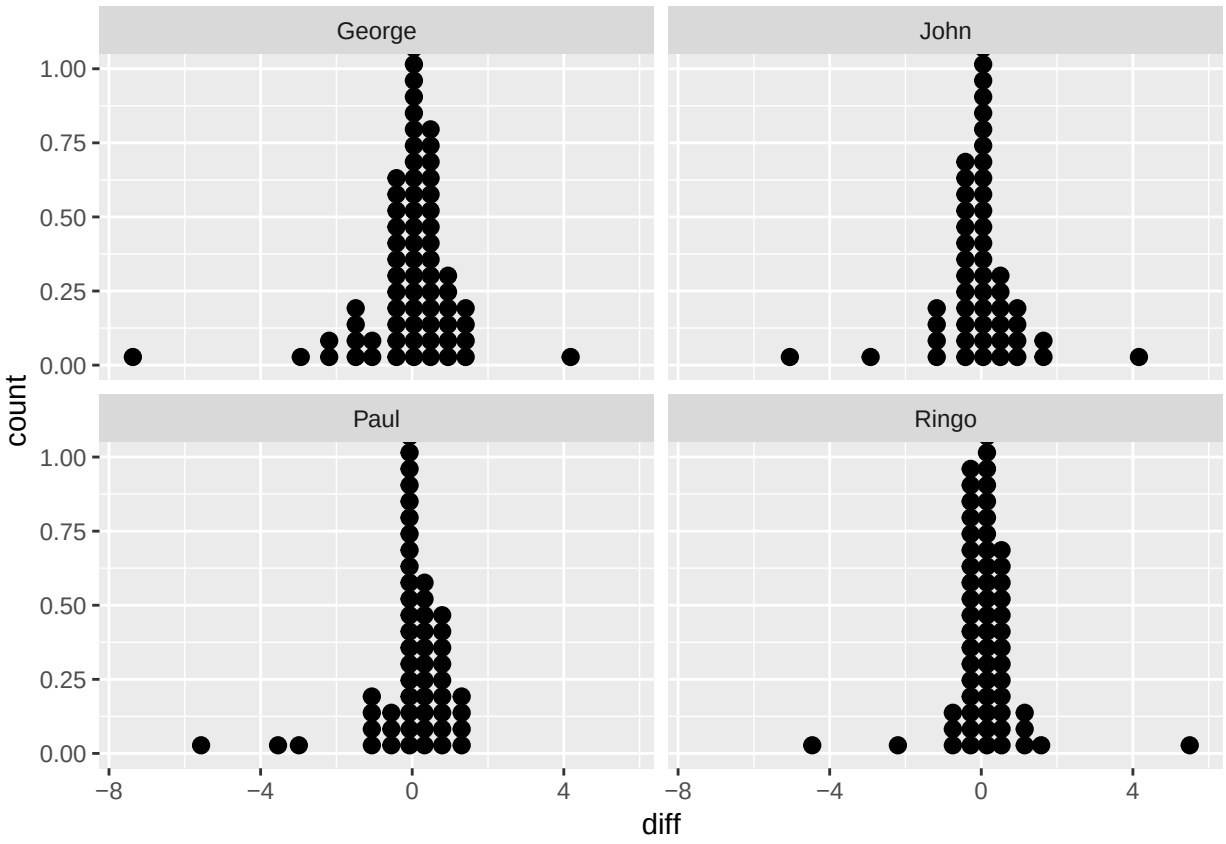



Differences are seen most clearly when plotted directly.

```
long %>%
  filter(method == "camera") %>%
  spread(test, value = value) %>%
  mutate(diff = a-b) %>%
  ggplot() +
  geom_dotplot(aes(x = diff)) +
  facet_wrap(~rater)
```

```
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 39 rows containing non-finite values (stat_bindot).
```




#### 3.5.6.3 Tidying the EU time-use-survey

```
library(lubridate)
library(eurostat)

# Eurostat table "tus_00selfstat"
eu_timeuse_complete <- get_eurostat('tus_00selfstat', type = 'label',
                                    stringsAsFactors = FALSE)

eu_timeuse <-
  eu_timeuse_complete %>%
  filter(sex == 'Total', wstatus == 'Population') %>%
  spread(unit, values) %>%
  mutate(year = year(time),
         prtcp_time_min = `Participation time (hh:mm)` %>%
           str_pad(width = 4, side = 'left', pad = '0') %>%
           {str_c(str_sub(., 1, 2), ':', str_sub(., 3, 4))} %>%
           hm() %>% time_length(unit = 'minutes'),
         time_spent_min = `Time spent (hh:mm)` %>%
           str_pad(width = 4, side = 'left', pad = '0') %>%
           {str_c(str_sub(., 1, 2), ':', str_sub(., 3, 4))} %>%
           hm() %>% time_length(unit = 'minutes')
         ) %>%
  select(activity = acl00, country = geo, year,
         prtcp_rate = `Participation rate (%)`,
```

```
         prtcp_time_min, time_spent_min)
```

```
## Warning in .parse_hms(..., order = "HM", quiet = quiet): Some strings
## failed to parse, or all strings are NAs
```

```
eu_timeuse
```

```
## # A tibble: 1,671 x 6
##   activity          country  year prtcp_rate prtcp_time_min time_spent_min
##   <chr>             <chr>   <dbl>      <dbl>          <dbl>          <dbl>
## 1 Activities relat~ Austria  2010       22.6             48             11
## 2 Activities relat~ Belgium  2000        1.4             88              1
## 3 Activities relat~ Belgium  2010       12.5             46              6
## 4 Activities relat~ Bulgar~  2000        7.1             39              3
## 5 Activities relat~ Estonia  2000       10.9             44              5
## # ... with 1,666 more rows
```

### 3.5.7 Excercise: Tidy data

- Recode the activity variable in dataset `eu_timeuse` into less than 10 meaningful categories of your own choice (make sure to filter out the "Total" values first). Visualize.

## 3.6 Joins

```
# population change
income <-
  get_eurostat('tgs00026', stringsAsFactors = FALSE) %>%
  select(geo, time, income = values)
```

```
## Table tgs00026 cached at /tmp/RtmpNiBgKC/eurostat/tgs00026_date_code_FF.rds
```

```
# unemployment rate
unemp <-
  get_eurostat('tgs00010', stringsAsFactors = FALSE) %>%
  filter(sex == 'T') %>%
  select(geo, time, unemp = values)
```

```
## Table tgs00010 cached at /tmp/RtmpNiBgKC/eurostat/tgs00010_date_code_FF.rds
```

```
# total fertility rate
totfert <-
  get_eurostat('tgs00100', stringsAsFactors = FALSE) %>%
  select(geo, time, totfert = values)
```

```
## Table tgs00100 cached at /tmp/RtmpNiBgKC/eurostat/tgs00100_date_code_FF.rds
```

```
# total life-expectancy
lifeexp <-
  get_eurostat('tgs00101', stringsAsFactors = FALSE) %>%
  filter(sex == 'T') %>%
  select(geo, time, lifeexp = values)
```

```
## Table tgs00101 cached at /tmp/RtmpNiBgKC/eurostat/tgs00101_date_code_FF.rds
```

```
# population change
popchange <-
  get_eurostat('tgs00099', stringsAsFactors = FALSE) %>%
  spread(indic_de, values) %>%
  select(geo, time,
         netmigrate = CNMIGRATRT,
         growthrate = GROWRT,
         natgrowthrate = NATGROWRT)
```

```
## Table tgs00099 cached at /tmp/RtmpNiBgKC/eurostat/tgs00099_date_code_FF.rds
```

```
#
eu_regional_indicators <-
  unemp %>%
  full_join(totfert) %>%
  full_join(lifeexp) %>%
  full_join(popchange) %>%
  full_join(income) %>%
  filter(str_length(geo) == 4) %>%
  mutate(country = str_sub(geo, end = 2)) %>%
  arrange(geo, time)
```

```
## Joining, by = c("geo", "time")
```

```
## Joining, by = c("geo", "time")
## Joining, by = c("geo", "time")
## Joining, by = c("geo", "time")
```

### 3.6.1 Excercise: Joins

- Create a dataset to a single topic of your choice by joining eurostat tables.

## 3.7 Tidy iteration

### 3.7.1 Group-wise operations

#### 3.7.1.1 Grouped `mutate()`

If we want to transform some columns in the data frame on a group-by-group basis we can use the `group_by()` together with `mutate()`.

Biologists sometimes express age not in years but in shares of total life-expectancy, i.e. the age of quarter life-expectancy, the age of half life-expectancy… Let's add this *relative* age to each life-table in the data. We need to

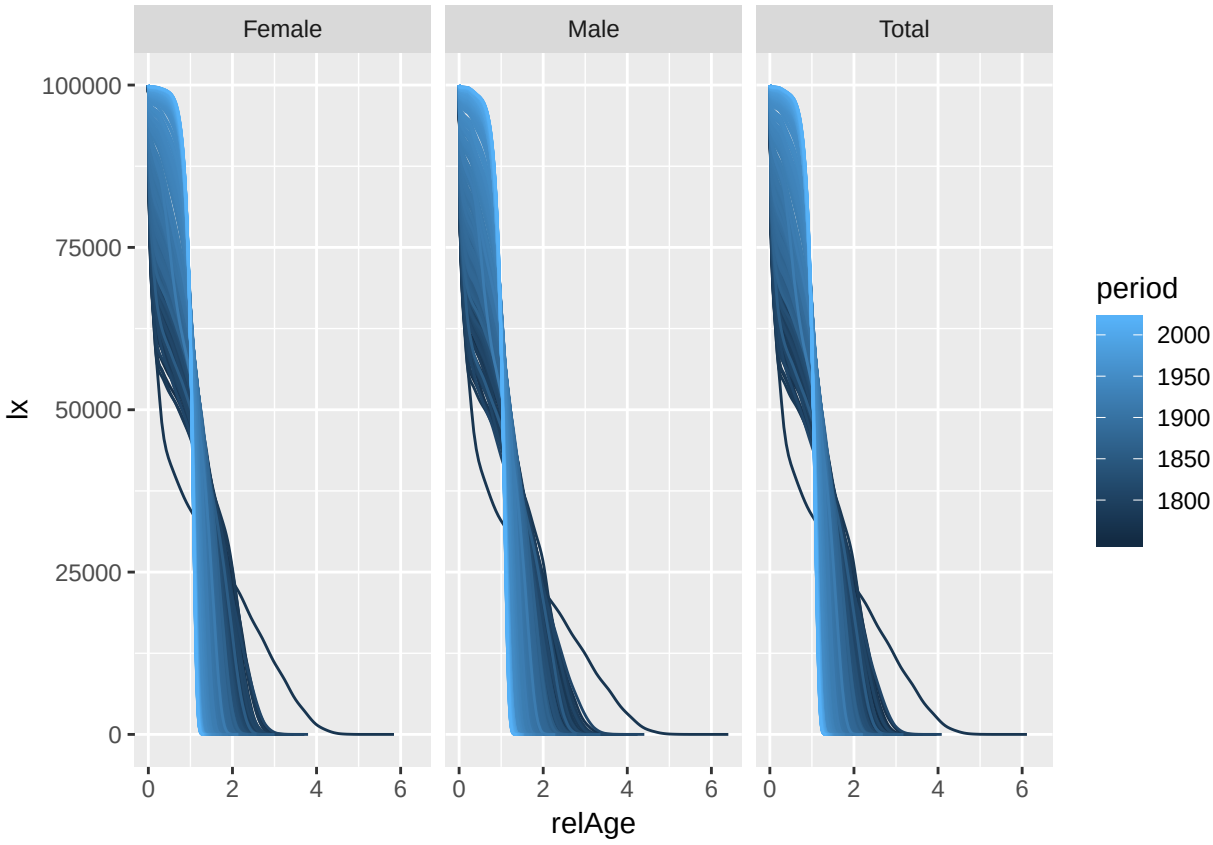
1. group our data into sub-groups defined by the values of country, sex and year
2. for each sub-group add a new column "relative age" to the life-table calculated as age over total life-expectancy
3. Re-combine the results of 2 into a data frame with columns identifying the sub-groups

```
hmd %>%
  group_by(country, sex, period) %>%
  mutate(relAge = age / ex[1]) %>%
  select(-(nx:ex))
```

```
## # A tibble: 1,304,694 x 6
## # Groups:   country, sex, period [11,754]
##   country country_name sex    period   age relAge
##   <chr>   <chr>        <chr>   <int> <int>  <dbl>
## 1 AUS     Australia    Female   1921     0 0
## 2 AUS     Australia    Female   1921     1 0.0158
## 3 AUS     Australia    Female   1921     2 0.0317
## 4 AUS     Australia    Female   1921     3 0.0475
## 5 AUS     Australia    Female   1921     4 0.0633
## # ... with 1.305e+06 more rows
```

Let's plot the life-table survivor function over relative age by sex for Sweden across periods.

```
hmd %>%
  group_by(country, sex, period) %>%
  mutate(relAge = age / ex[1]) %>%
  ungroup() %>%
  filter(country == 'SWE') %>%
  ggplot() +
  geom_line(aes(x = relAge, y = lx, group = period, color = period)) +
  facet_wrap(~sex)
```



#### 3.7.1.2 Grouped `filter()`

#### 3.7.1.3 Grouped `slice()`

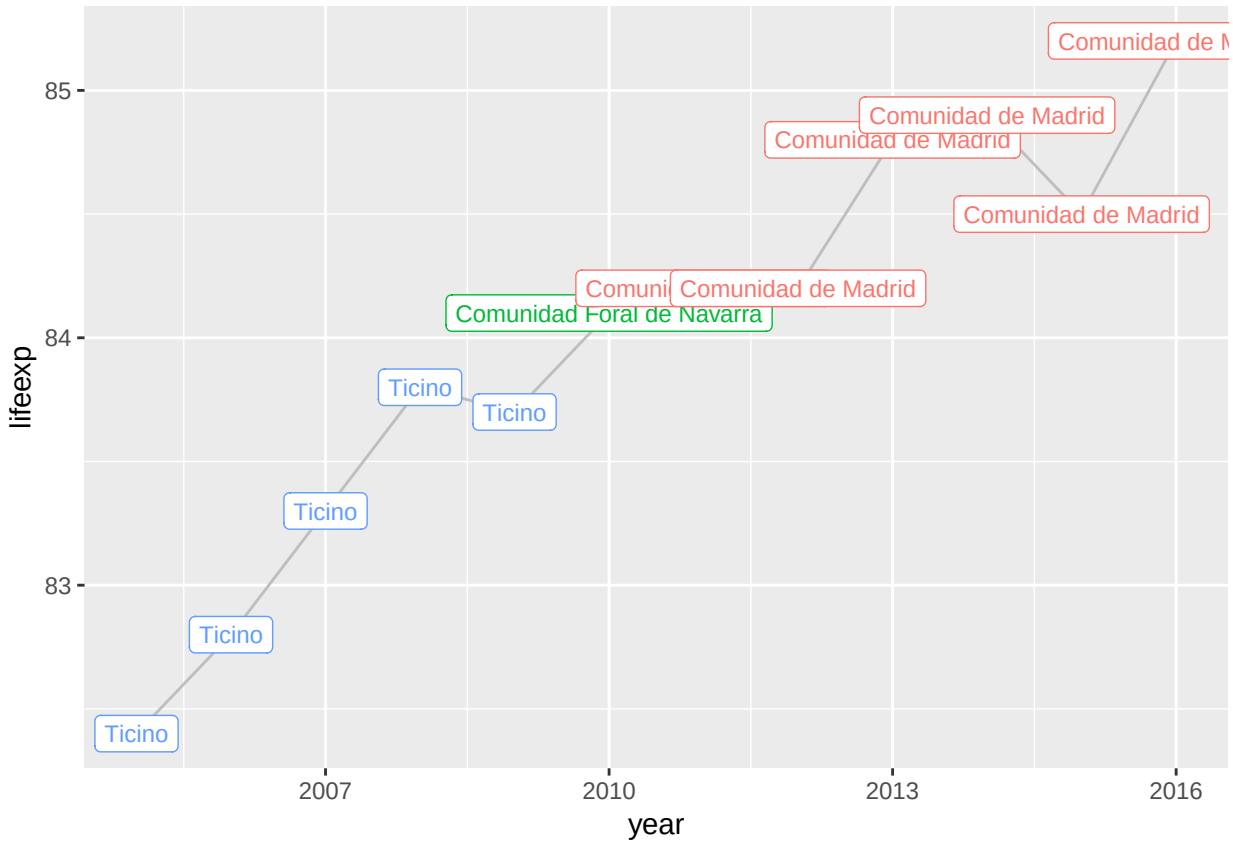
`slice()` selects rows by index. Using `slice()` in a grouped fashion we can return the first and last row for each group – this comes in handy as a quick check for data structure and coverage.

```
euro_regio %>%
  # for each country...
  group_by(country_name) %>%
  # ...return the first and last row
  slice(c(1, n()))
```

```
## # A tibble: 72 x 21
## # Groups:   country_name [36]
##   country_code country_name nuts2_code nuts2_name       year   pop popdens
##   <chr>        <chr>        <chr>      <chr>           <int> <dbl>   <dbl>
## 1 AL           Albania      AL01       Veri             2005    NA    NA
## 2 AL           Albania      ALXX       ALXX Not regio~  2017    NA    NA
## 3 AT           Austria      AT11       Burgenland (AT)  2005    NA    75.7
## 4 AT           Austria      ATZZ       ATZZ Extra-Reg~  2017    NA    NA
## 5 BE           Belgium      BE10       Région de Brux~  2005    NA  6290.
## # ... with 67 more rows, and 14 more variables: births <dbl>,
## #   deaths <dbl>, natgrowthrate <dbl>, growthrate <dbl>, netmigrate <dbl>,
## #   ...
```

We can use grouped `slice()` to have a closer look at outliers within groups. This replicates the Oeppen-Vaupel line for European regions.

```
euro_regio %>%
  # for each year
  group_by(year) %>%
  #...select the row (region) with the highest life-expectancy
  slice(which.max(lifeexp)) %>%
  # ...and plot the results
  ggplot() +
  geom_line(aes(x = year, y = lifeexp), color = 'grey') +
  geom_label(aes(x = year, y = lifeexp,
                 label = nuts2_name, color = nuts2_name),
             show.legend = FALSE, size = 3)
```




#### 3.7.1.4 Grouped `summarise()`

Say we have a collection of life-tables by country, sex, and year and we want to calculate the coefficient of variation for the life-table distribution of deaths. In other words we want to

1. group our data into subgroups defined by the values of country, sex and year (so a single sub-group may be Danish females in 2010)
2. extract total life-expectancy from each sub-group life-table
3. calculate the coefficient of variation for each sub-group
4. Re-combine the results of 2 and 3 into a data frame with columns identifying the sub-groups

All of the above is achieved by the data pipeline below.

```
hmd %>%
  group_by(country, sex, period) %>%
  summarise(
    e0 = first(ex),
    cv = sqrt(sum(ndx/100000*(age+nax-e0)^2)) / e0
  )
```

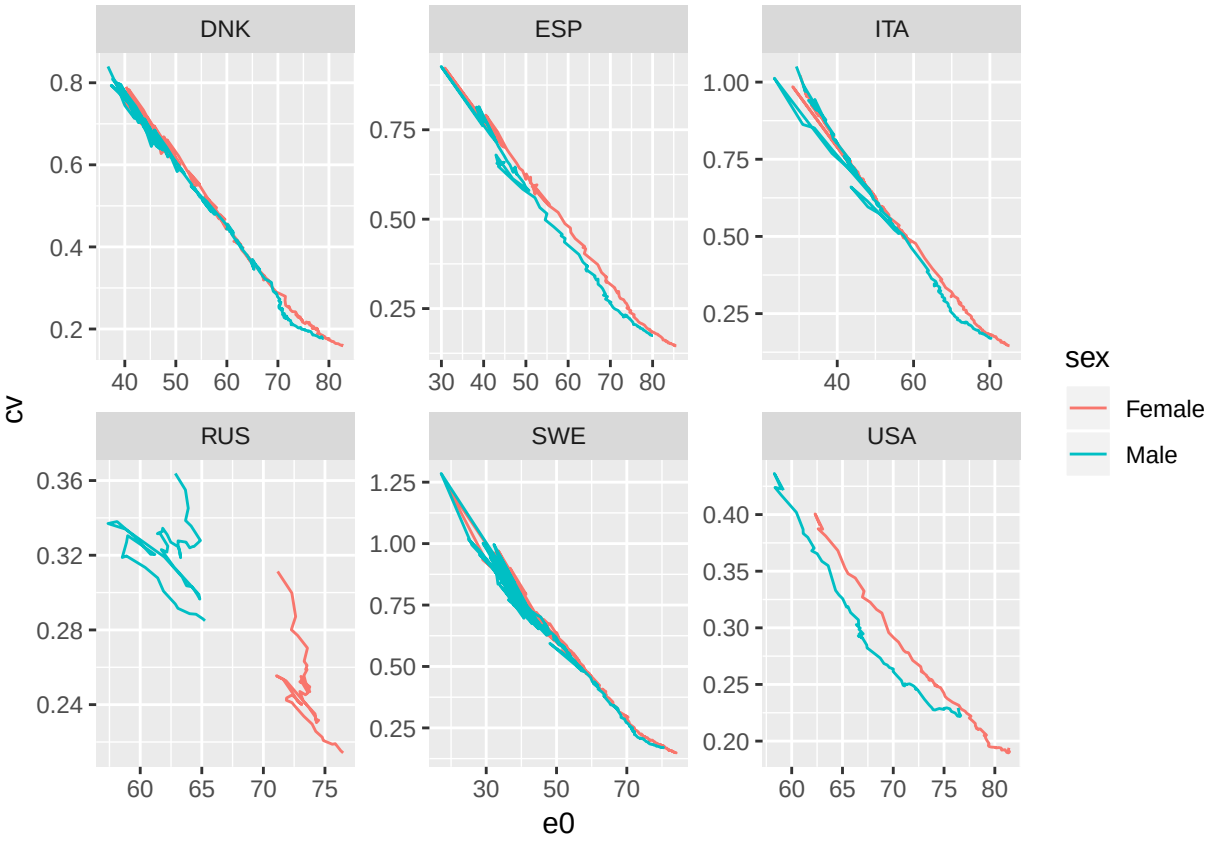
```
## # A tibble: 11,754 x 5
## # Groups:   country, sex [?]
##   country sex    period    e0    cv
##   <chr>   <chr>   <int> <dbl> <dbl>
## 1 AUS     Female   1921  63.2 0.409
## 2 AUS     Female   1922  65.1 0.370
## 3 AUS     Female   1923  63.7 0.388
## 4 AUS     Female   1924  64.5 0.383
```

```
## 5 AUS     Female   1925  65.4 0.370
## # ... with 1.175e+04 more rows
```

We use the `group_by()` function to group our data into sub-groups, then we use the `summarise()` command to calculate the "summary statistics" for each sub-group. The `ungroup()` function in the end is optional but its good practive to ungroup after you're done with the group-wise operations.

Let's plot the results (for a subset of all countries):

```
hmd %>%
  filter(sex != 'Total') %>%
  group_by(country, sex, period) %>%
  summarise(
    e0 = first(ex),
    cv = sqrt(sum(ndx/100000*(age+nax-e0)^2)) / e0
  ) %>% ungroup() %>%
  filter(country %in% c('SWE', 'RUS', 'ITA', 'DNK', 'USA', 'ESP')) %>%
  ggplot() +
  geom_path(aes(x = e0, y = cv, color = sex)) +
  facet_wrap(~country, scales = 'free')
```




#### 3.7.1.5 Grouped `do()`

#### 3.7.1.6 Fitting and summarising many models

#### 3.7.1.7 Excercise: Group-wise operations

- Calculate five year abridged mortality rates for the whole `hmd_counts` data.

### 3.7.2 Column-wise operations

#### 3.7.2.1 *_at()

```
# apply function "toupper()" to the columns country, sex
mutate_at(hmd_counts, vars(country, sex), toupper)
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx   nDx    nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     FEMALE   1921     0     1 3842. 64052.
## 2 AUS     FEMALE   1921     1     1  719. 59619.
## 3 AUS     FEMALE   1921     2     1  330. 57126.
## 4 AUS     FEMALE   1921     3     1  166. 57484.
## 5 AUS     FEMALE   1921     4     1  190. 58407.
## # ... with 1.305e+06 more rows
```

```
# this returns the same as above
mutate(hmd_counts,
       country = toupper(country),
       sex = toupper(sex))
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx   nDx    nEx
##   <chr>   <chr>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     FEMALE   1921     0     1 3842. 64052.
## 2 AUS     FEMALE   1921     1     1  719. 59619.
## 3 AUS     FEMALE   1921     2     1  330. 57126.
## 4 AUS     FEMALE   1921     3     1  166. 57484.
## 5 AUS     FEMALE   1921     4     1  190. 58407.
## # ... with 1.305e+06 more rows
```

#### 3.7.2.2 *_if()

```
# convert any character variable to a factor
mutate_if(hmd_counts, is.character, as.factor)
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx   nDx    nEx
##   <fct>   <fct>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     Female   1921     0     1 3842. 64052.
## 2 AUS     Female   1921     1     1  719. 59619.
## 3 AUS     Female   1921     2     1  330. 57126.
## 4 AUS     Female   1921     3     1  166. 57484.
## 5 AUS     Female   1921     4     1  190. 58407.
## # ... with 1.305e+06 more rows
```

```
# this returns the same as above
mutate(hmd_counts,
       country = as.factor(country),
       sex = as.factor(sex))
```

```
## # A tibble: 1,304,694 x 7
##   country sex    period   age    nx   nDx    nEx
```

```
##   <fct>   <fct>   <int> <int> <int> <dbl>  <dbl>
## 1 AUS     Female   1921     0     1 3842. 64052.
## 2 AUS     Female   1921     1     1  719. 59619.
## 3 AUS     Female   1921     2     1  330. 57126.
## 4 AUS     Female   1921     3     1  166. 57484.
## 5 AUS     Female   1921     4     1  190. 58407.
## # ... with 1.305e+06 more rows
```

#### 3.7.2.3 `*_all()`

#### 3.7.2.4 Excercise: Column-wise operations

- Download `get_eurostat('t2020_10', time_format = 'raw')` and write a pipe that calculates the difference between the current value and the target value over time for each country.

# Chapter 4

# Visualization

## 4.1 The "tidy" approach to data visualization

Data visualization in the `tidyverse` revolves around three concepts:

1) **Tidy Data** Data is stored is a data frame with cases as rows and variables as columns.
2) **Mappings** Variables are mapped to visual attributes, called *aesthetics*.
3) **Layers** Plots are build layer by layer. There are different types of layers, each changing a different aspect of a plot, i.e. coordinate system, scales, statistical transformations or visual marks.

Part of the tidyverse is a package called `ggplot2` which is what we are going to use to produce plots. `ggplot2` is an implementation of the *Grammar of Graphics*, a domain specific language to specify visualizations.

## 4.2 The Grammar of graphics

*Data*, *Statistics*, *Aestetics*, *Scales*, *Geometry*, *Coordinates*.

**Data** Observations of one or more variables.

**Statistics** Transformations of the data prior to plotting.

- Algebraic transformations (log, exp, sqrt, identical…)
- Aggregating statistical transformations (mean, median, mode, bin, linear regression…)
- Non-aggregating statistical transformations (sort, rank, cut…)

**Aesthetics** The visual representation of a variable.

- Position. `x`, `y`, `xmin`, `xmax`, `ymin`, `ymax`, `xend`, `yend`
- Colour. `colour`, `fill`, `alpha`
- Size. `size`
- Shape, Pattern. `shape`, `linetype`
- Group. `group`
- Angle, Slope, Intercept, Texture, …

**Scales** The way a variable is mapped to a visual representation.

- Discrete, Continous. e.g. `scale_x_discrete`, `scale_colour_continuous`
- Concerning Position, Colour, Size, Shape… e.g. `scale_y_continuous`, `scale_size`, `scale_shape()`
- Breakpoints, Labels
- Transformations (log, sqrt, reverse)


Figure 4.1: Gapminder World

**Geometry** General "shape" or "type" of the graphic

- Scatterplots use `geom_point`,
- Timelines use `geom_line`,
- Area charts use `geom_area`,
- Heatmaps use `geom_tile`,
- Bar charts use `geom_bar`,
- there are many more…

**Coordinates** The coordinate system of the plot area.

- Cartesian coordinates
- Polar coordinates

### 4.2.1 Aesthetics, geometries, and layers

Interactive plots displayed in a web-browser have long arrived in the mainstream with Gapminder World being a true classic of the genre.

Today we shall recreate the above chart and in doing so learn the basics of ggplot. Key concepts we will tackle are *aesthetics*, *layers*, *scales*, and *facets*.

Each ggplot starts with *data*. **Data must be a data frame**! Today we will use data which comes in form of an R package: The gapminder data.

```
library(tidyverse)

library(gapminder)
is.data.frame(gapminder)
```

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

```
head(gapminder)
```

```
## # A tibble: 6 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
## 5 Afghanistan Asia       1972    36.1 13079460      740.
## # ... with 1 more row
```

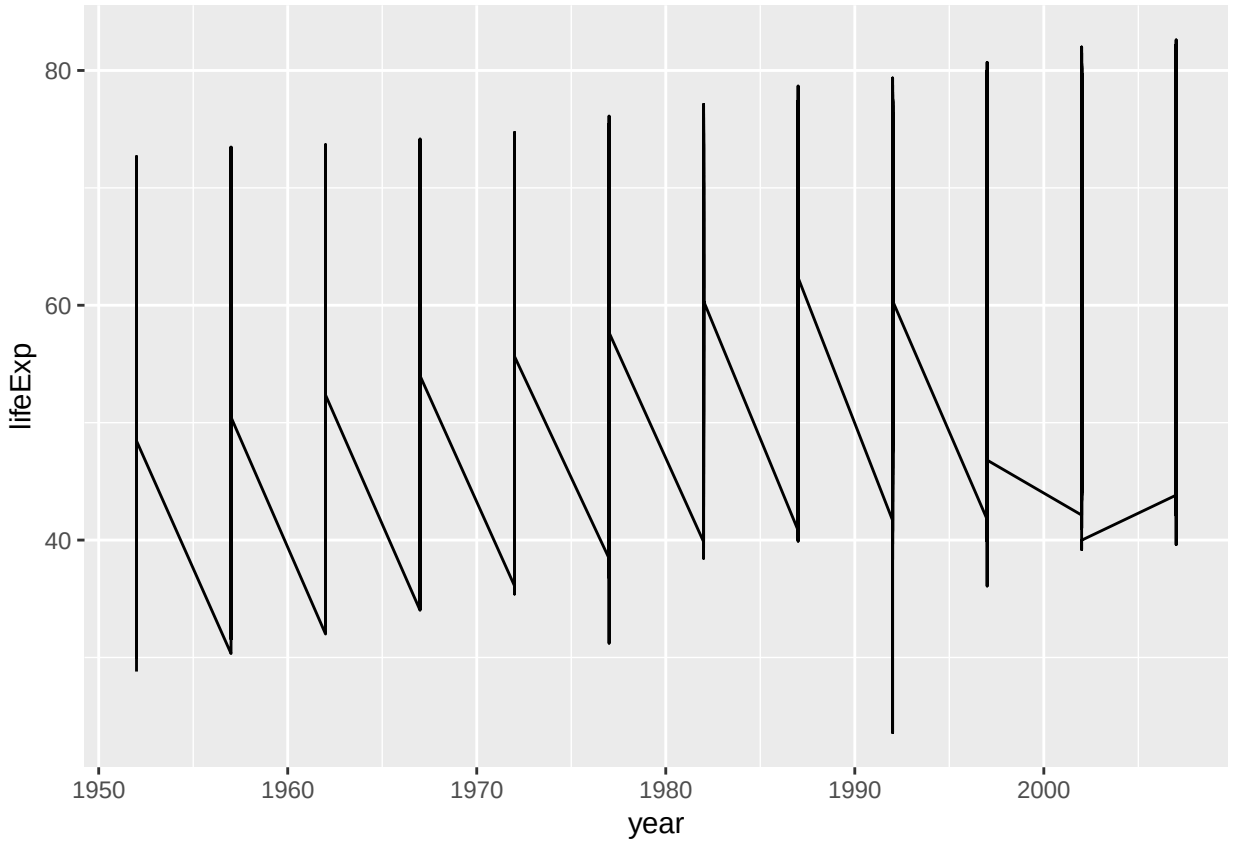
First step in plotting with ggplot – provide ggplot with a data frame:

```
ggplot(data = gapminder)
```



ggplot knows about our data but nothing happens yet. We need to add a *layer*. **We add elements to a plot by adding them with a +.**

```
ggplot(data = gapminder) +
  geom_line(aes(x = year, y = lifeExp))
```

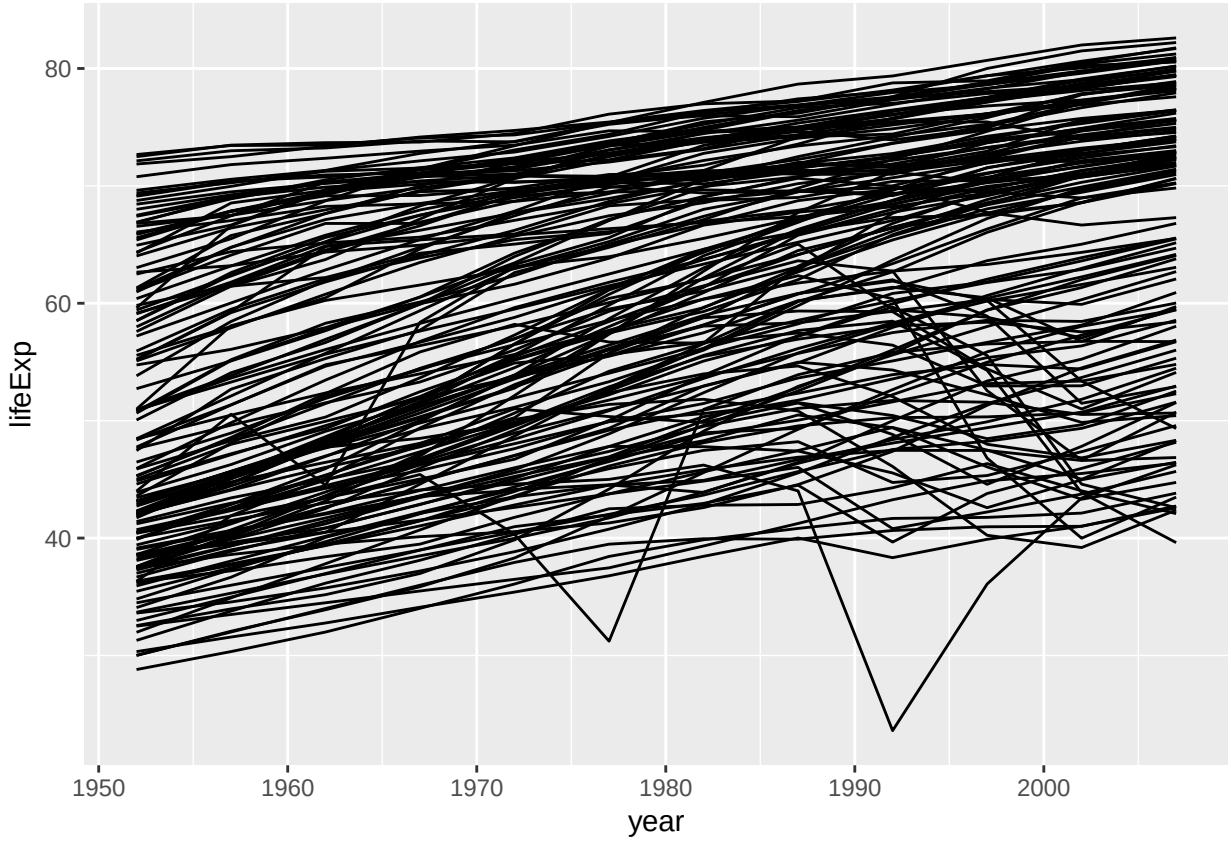



`aes` stands for *aesthetics*: Mappings between variables in our data and visual properties. Each column in our data frame is a variable. Visual properties are manifold and can be `x`, `y`, `colour`, `size`, `shape`, `alpha`, `fill`, `radius`, `linetype`, `group`…

We use the `aes()` function to map `x`-position to the variable `Time` and `y`-position to the variable `weight`. **Every time you map a variable to a visual property you do it inside `aes()`**

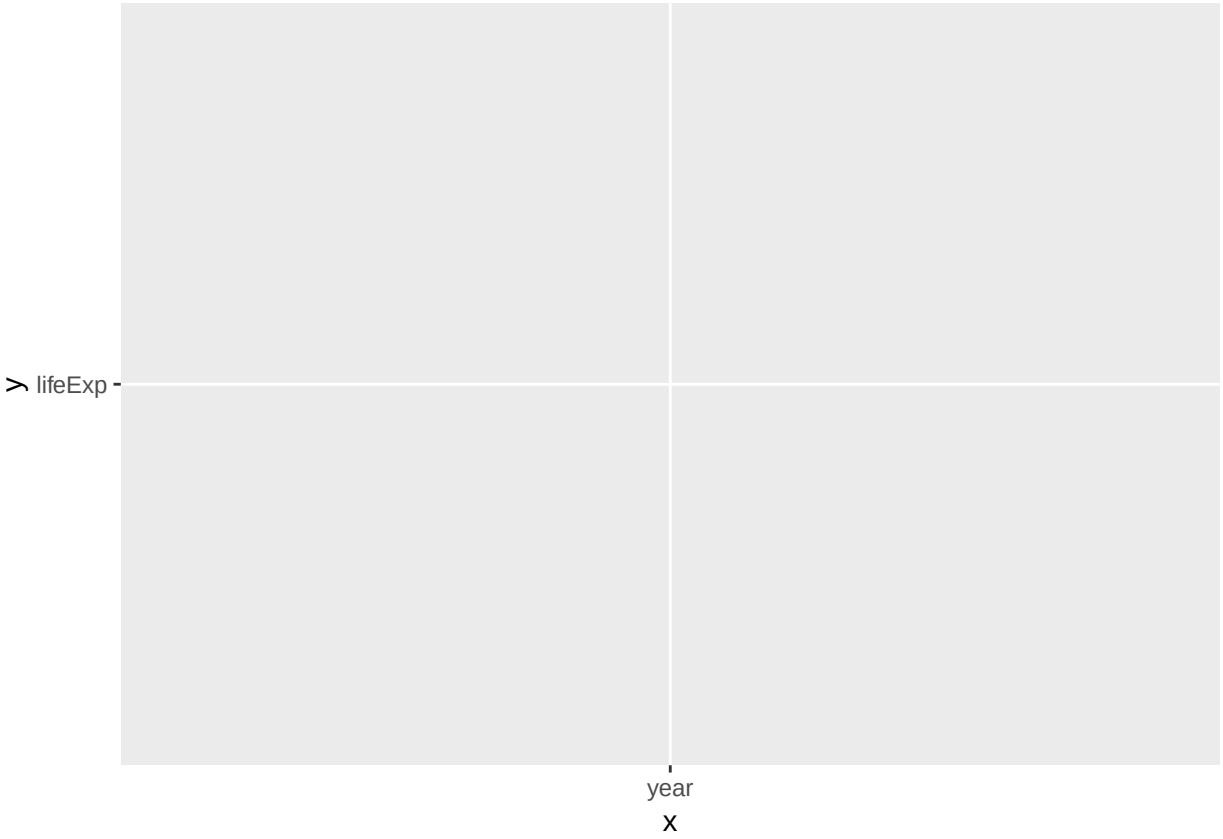
The plot looks strange. A single line is drawn across all data points. We want separate lines – one time series per country…

```
ggplot(data = gapminder) +
  geom_line(aes(x = year, y = lifeExp, group = country))
```




Note that we don't write `x = gapminder$year` or `y = "lifeExp"`. We simply spell out the name of the variable we wish to work with. ggplot is aware of the dataset we work with – it is *attached*. Quoting the variable names would actually produce unexpected results:

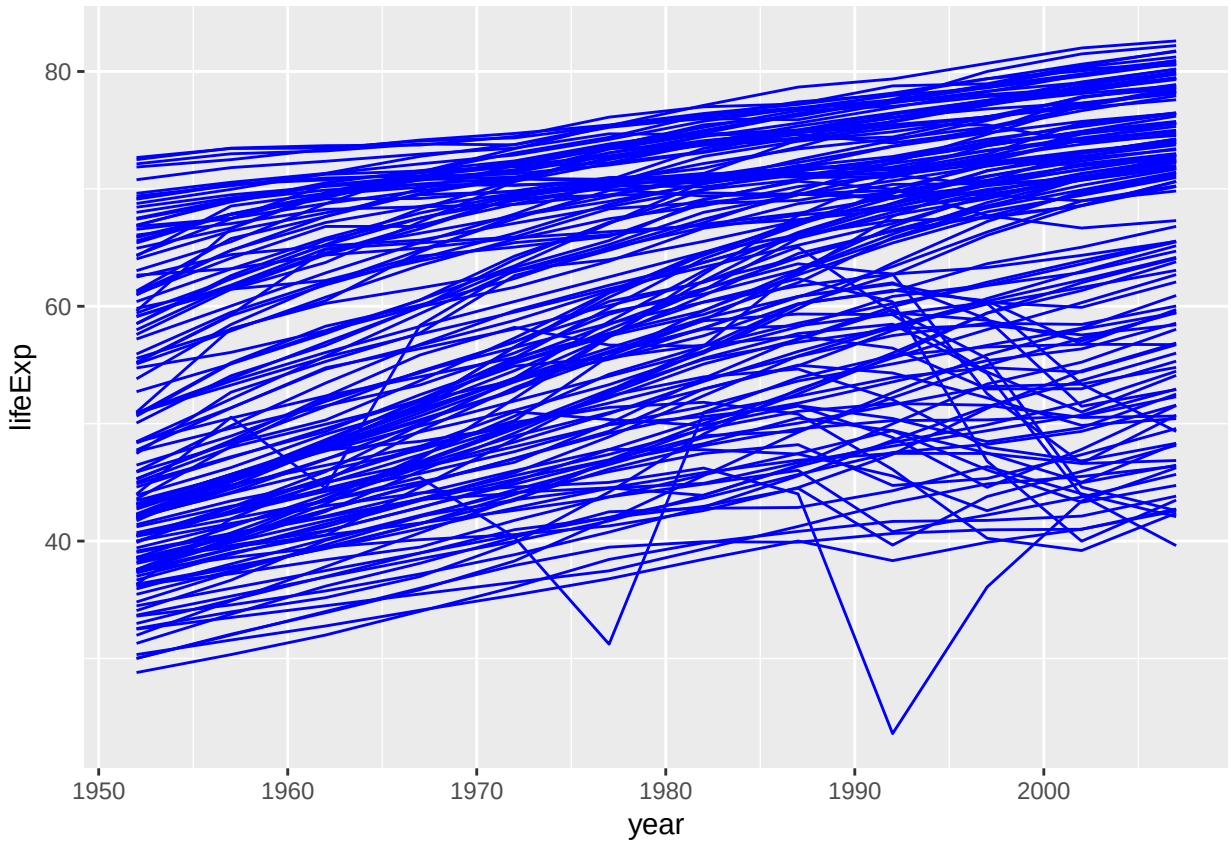
```
ggplot(data = gapminder) +
  geom_line(aes(x = "year", y = "lifeExp", group = "country"))
```




What happened? ggplot interpreted the quoted strings as raw data instead of variable names of our data frame. It then tries to plot it… **Always use unquoted column names to address the variables in your data.**

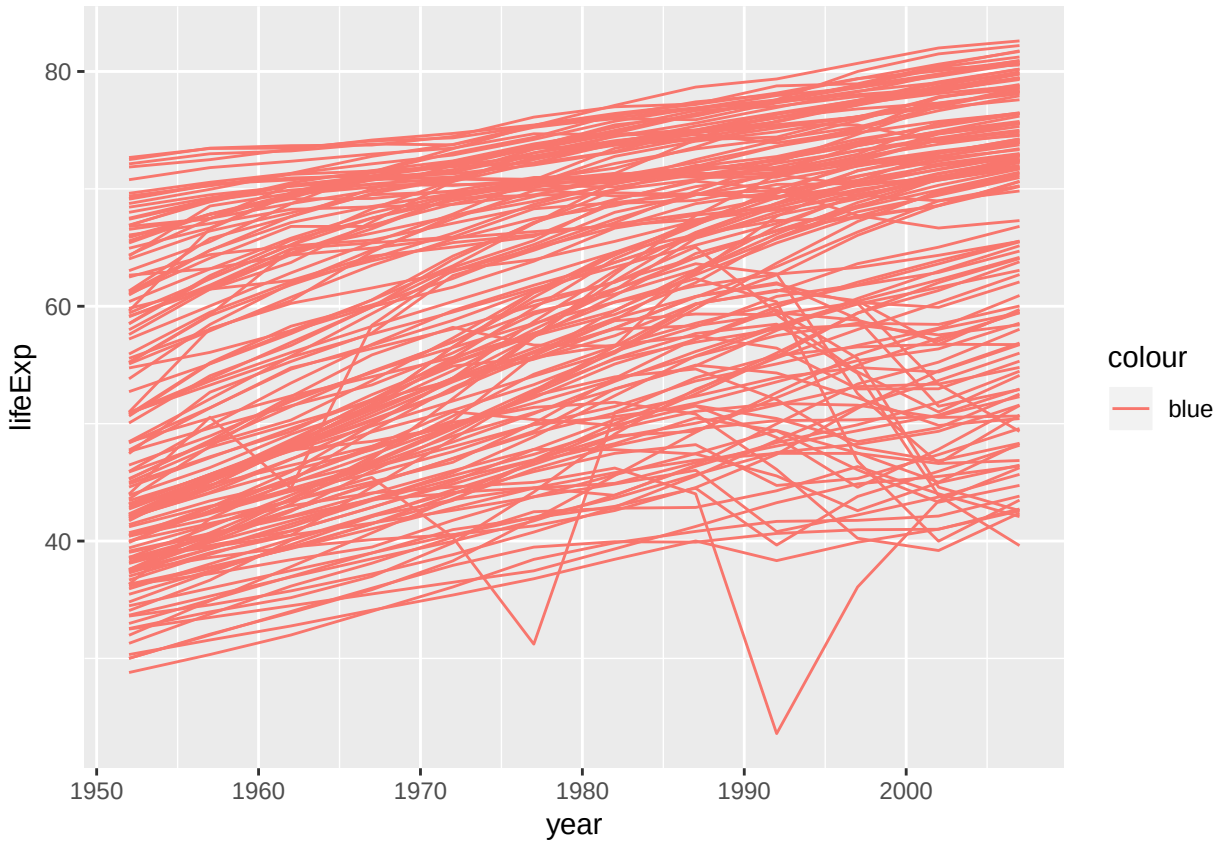
Let's colour all of the lines blue.

```
ggplot(data = gapminder) +
  geom_line(aes(x = year, y = lifeExp, group = country), colour = "blue")
```



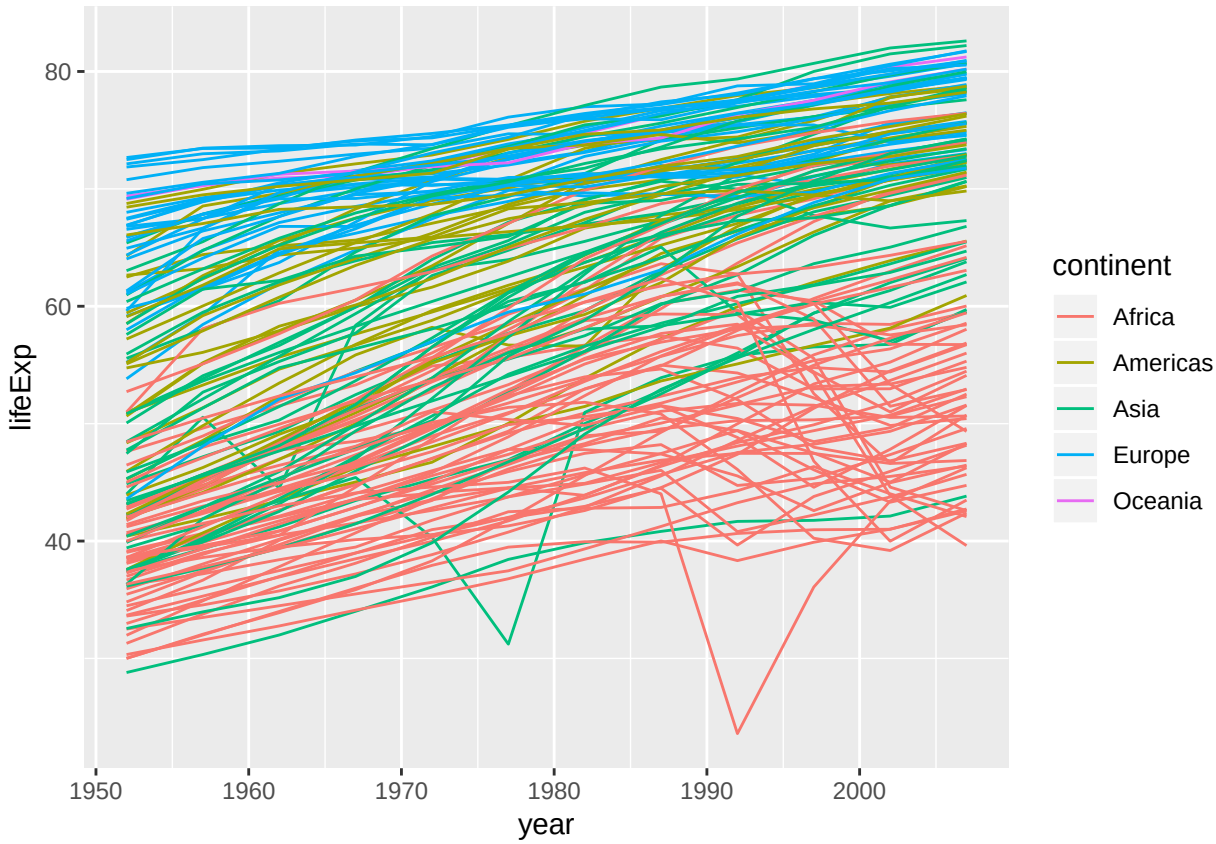

We wrote `colour = "blue"` outside of the `aes()` function as we *set* the visual property to a fixed value instead of *mapping* a visual property to a variable in the data frame. For comparison, let's move the colour specification into the `aes()` function:

```
ggplot(data = gapminder) +
  geom_line(aes(x = year, y = lifeExp, group = country, colour = "blue"))
```




Classic ggplot moment here… "blue" gets interpreted as raw data as we have written it inside of the `aes()` function. ggplot thinks all of our rows belong to group "blue", mapped to the visual property colour. ggplot assigns a default colour scale of which the first colour is a light red. Here's the same behaviour, but this time it makes sense as we map colour to an actual variable in our data set.

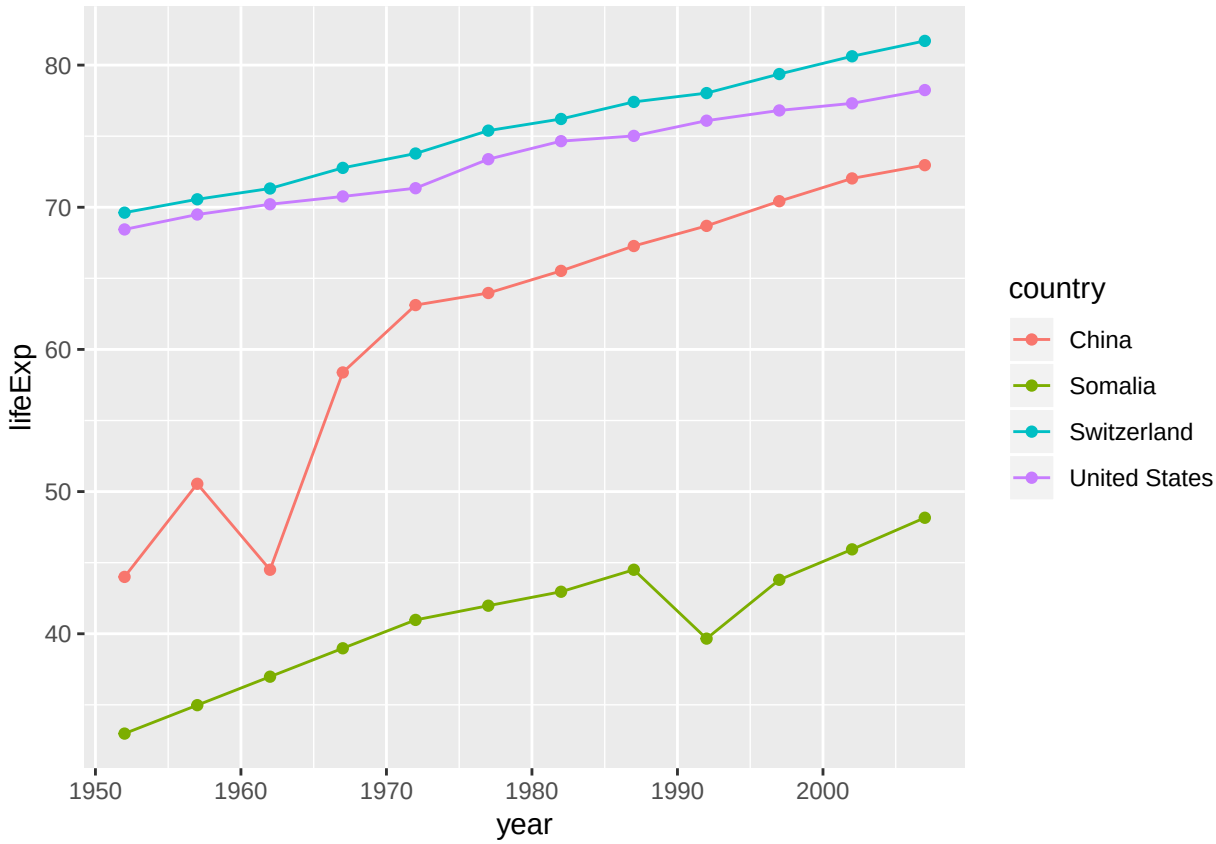
```
ggplot(data = gapminder) +
  geom_line(aes(x = year, y = lifeExp, group = country, colour = continent))
```




Lesson to be learned: The difference between *mapping* and *setting* a visual property. **You map visual properties to variables inside `aes()`, you set visual properties to a fixed value outside of `aes()`.**

We can add more than a single layer to a plot. ggplot 2.0.0 comes with 27 *geometries* (`geom`). Some of them are super straightforward and just draw points or lines or rectangles. Some draw complex shapes after transforming your data in various ways. **Think of geometries as flexible templates for different plot types.** Combining different geometries is a common workflow in ggplot. E.g. adding `geom_point` to `geom_line` gives lines with points on them:

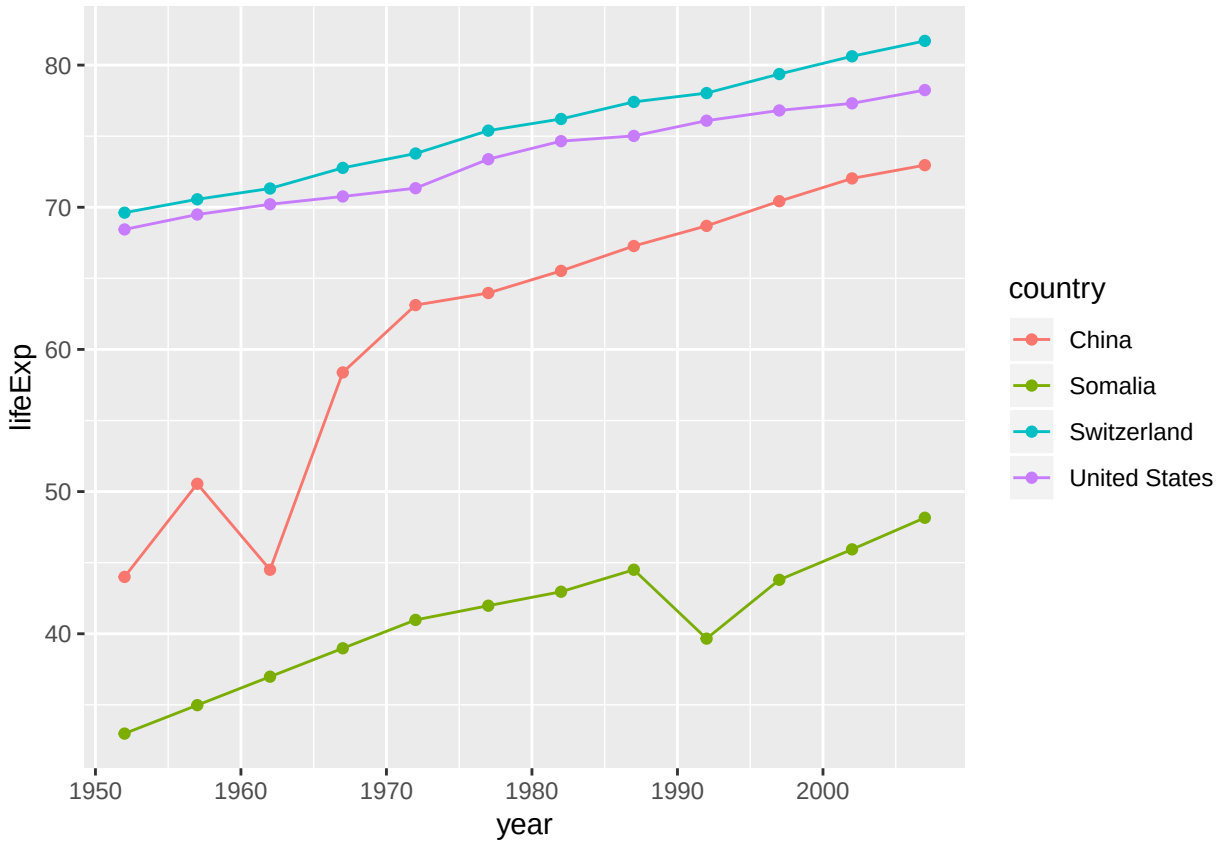
```
# filter data to 4 countries only
gapminder_sub <-
  filter(gapminder,
         country %in% c("China", "Switzerland", "United States", "Somalia"))

ggplot(data = gapminder_sub) +
  geom_line(aes(x = year, y = lifeExp, color = country)) +
  geom_point(aes(x = year, y = lifeExp, color = country))
```




Note that we did not need to specify the `group` argument in the last ggplot call. The have mapped colour to country and therefore implicitly specified the grouping structure of our data. We use `group` only if ggplot fails to correctly guess the grouping.

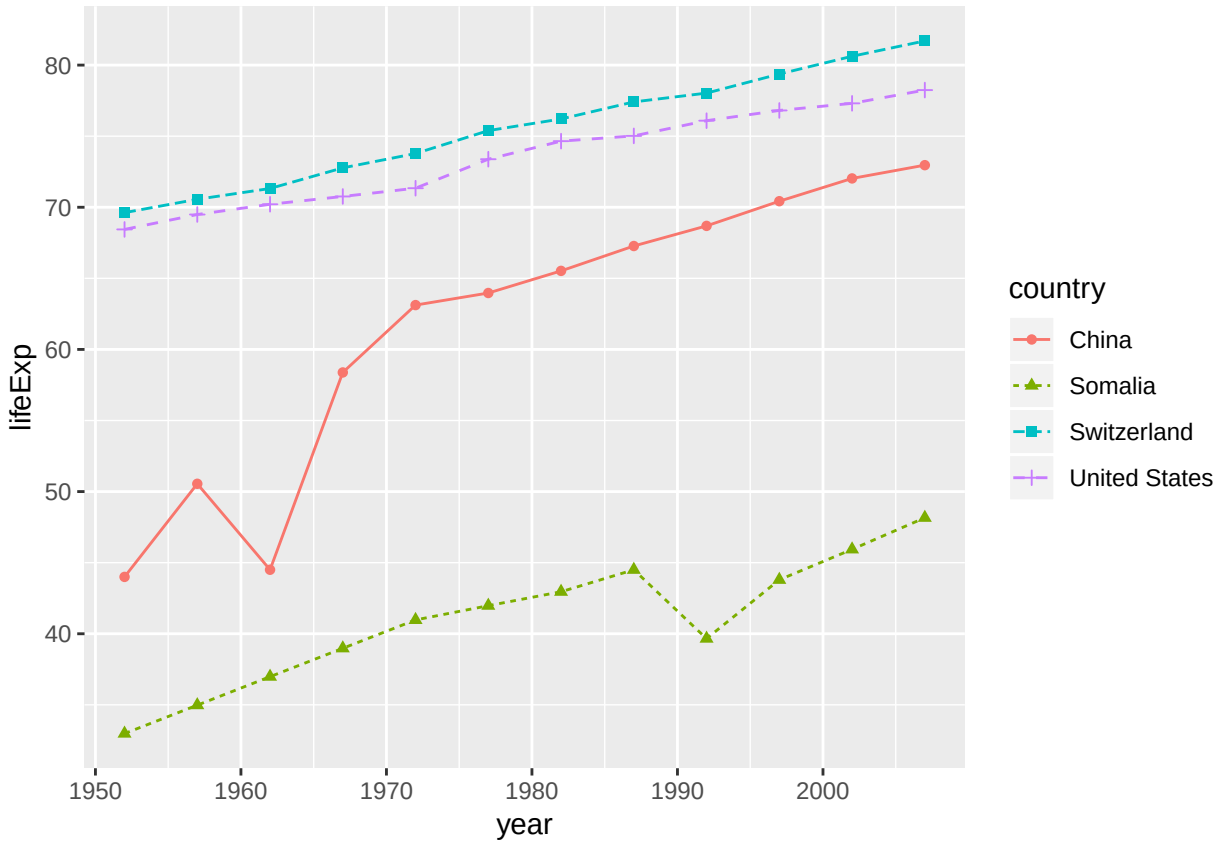
If we use identical mappings in our layers we can move them into the `ggplot()` function. **Everything inside the `ggplot()` function is passed down to all other plot elements.**

```
ggplot(data = gapminder_sub, aes(x = year, y = lifeExp, color = country)) +
  geom_line() +
  geom_point()
```



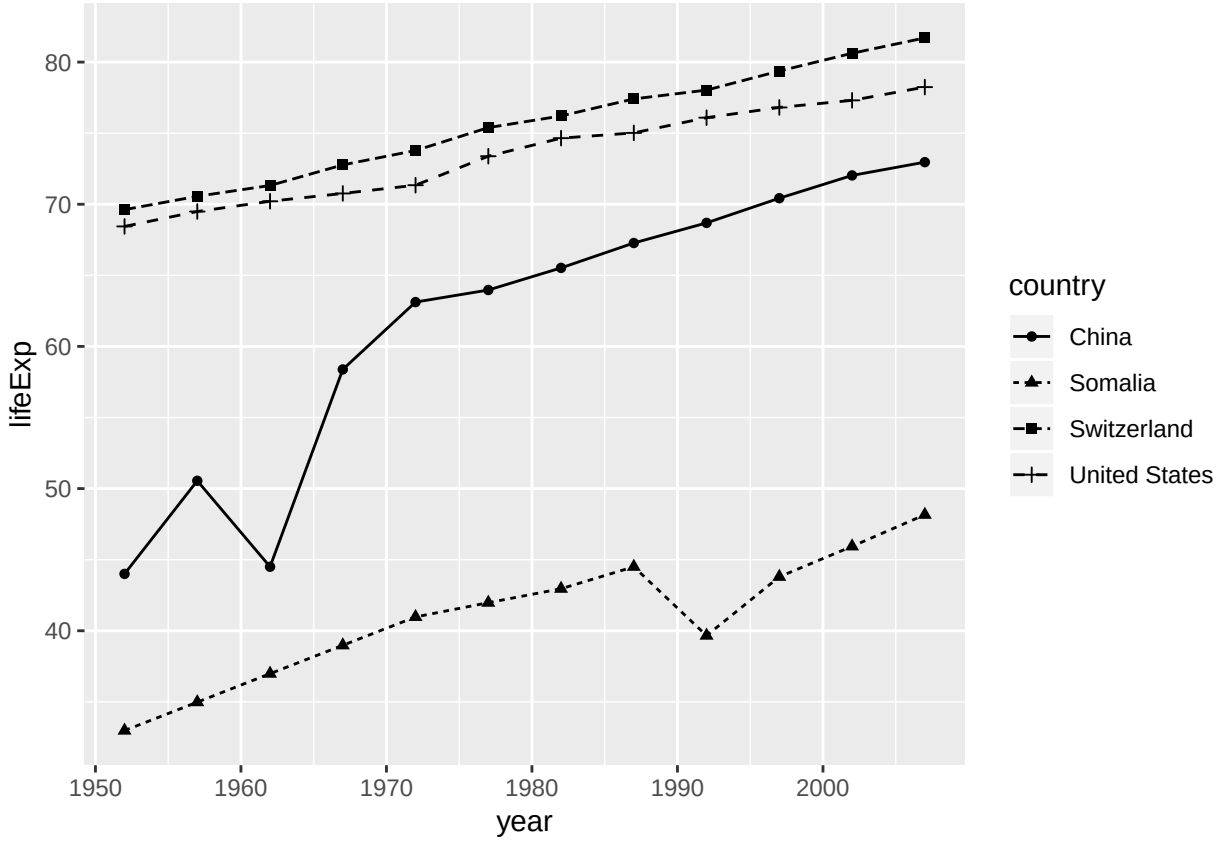

We can still add arguments to the individual layers.

```
ggplot(data = gapminder_sub, aes(x = year, y = lifeExp, color = country)) +
  geom_line(aes(linetype = country)) +
  geom_point(aes(shape = country))
```




We also have the power to override commands from above:

```
ggplot(data = gapminder_sub, aes(x = year, y = lifeExp, color = country)) +
  geom_line(aes(linetype = country), colour = "black") +
  geom_point(aes(shape = country), colour = "black")
```




### 4.2.2 Scales

We already know that *aesthetics* are mappings from data dimensions to visual properties. They tell ggplot what goes where. What are the aesthetics in **Gapminder World**?

| Data dimension | Visual property | Scale |
|---|---|---|
| GDP per capita | position on x-axis (`x`) | `scale_x_*` |
| Life expectancy | position on y-axis (`y`) | `scale_y_*` |
| Population size | size of plotting symbols (`size`) | `scale_size_*` |
| Geographical Region | colour of plotting symbols (`colour`) | `scale_colour_*` |

**Each aesthetic has its own scale**

The four aesthetics in *Gapminder World* are connected to four different scales. The scales are named after the corresponding aesthetic. The naming scheme is `scale_<name of aesthetic>_<continuous|discrete|specialized>`.

**Aesthetics specify the *what*, scales specify the *how***

Which colour to use for which level in the data? Where to put the labels on the axis? Which labels to put? The size of the largest plotting symbol, the name of the legends, log-transformation of the y-axis, the range of the axis… These are all examples of scale specifications – specifications on *how* to map a data dimension to a visual attribute.

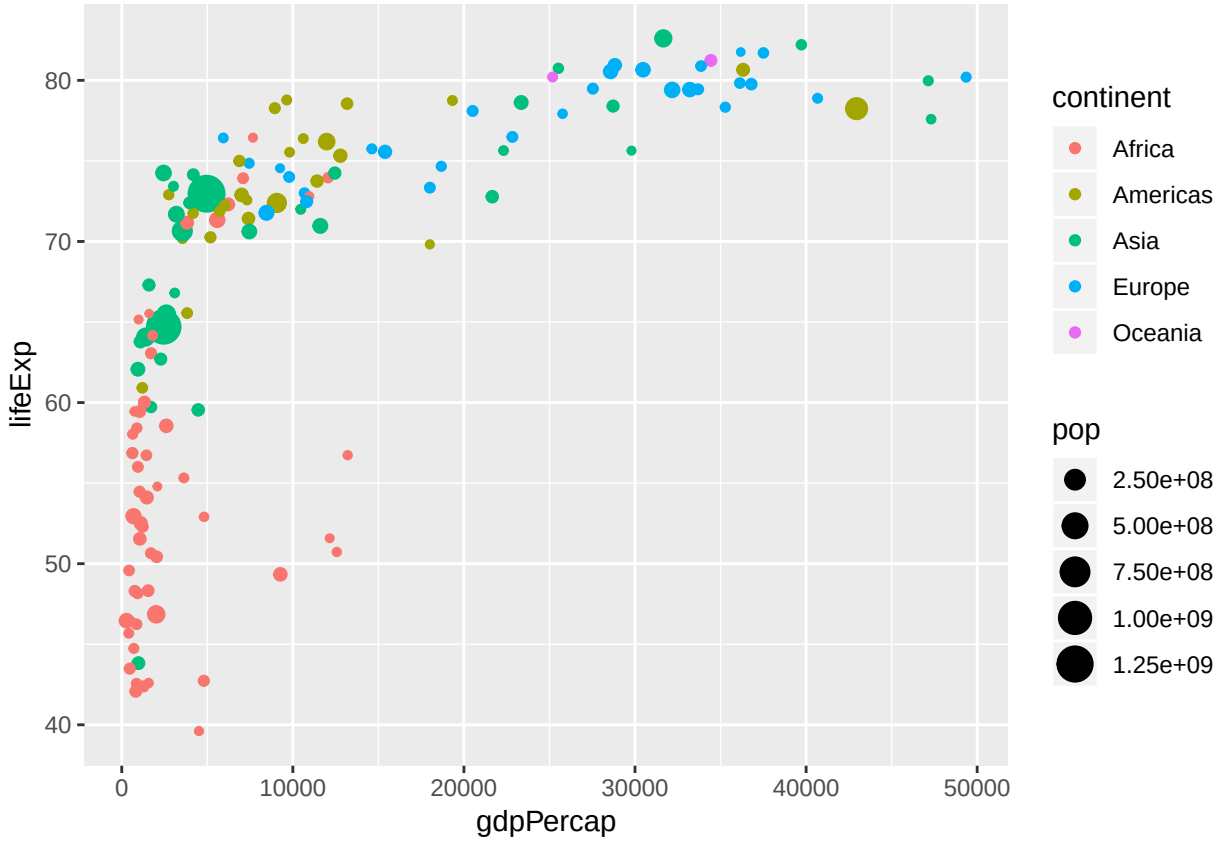
Off to work!

```
library(gapminder)
head(gapminder)
```

```
## # A tibble: 6 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
## 5 Afghanistan Asia       1972    36.1 13079460      740.
## # ... with 1 more row
```

The data already looks tidy. All we have to do is to subset to a single year. Let's see what ggplot produces if we simply specify the aesthetics to an appropriate geometry.

```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point()
```




A solid foundation. But to close in on the *Gapminder World* chart we need to customize our scales.

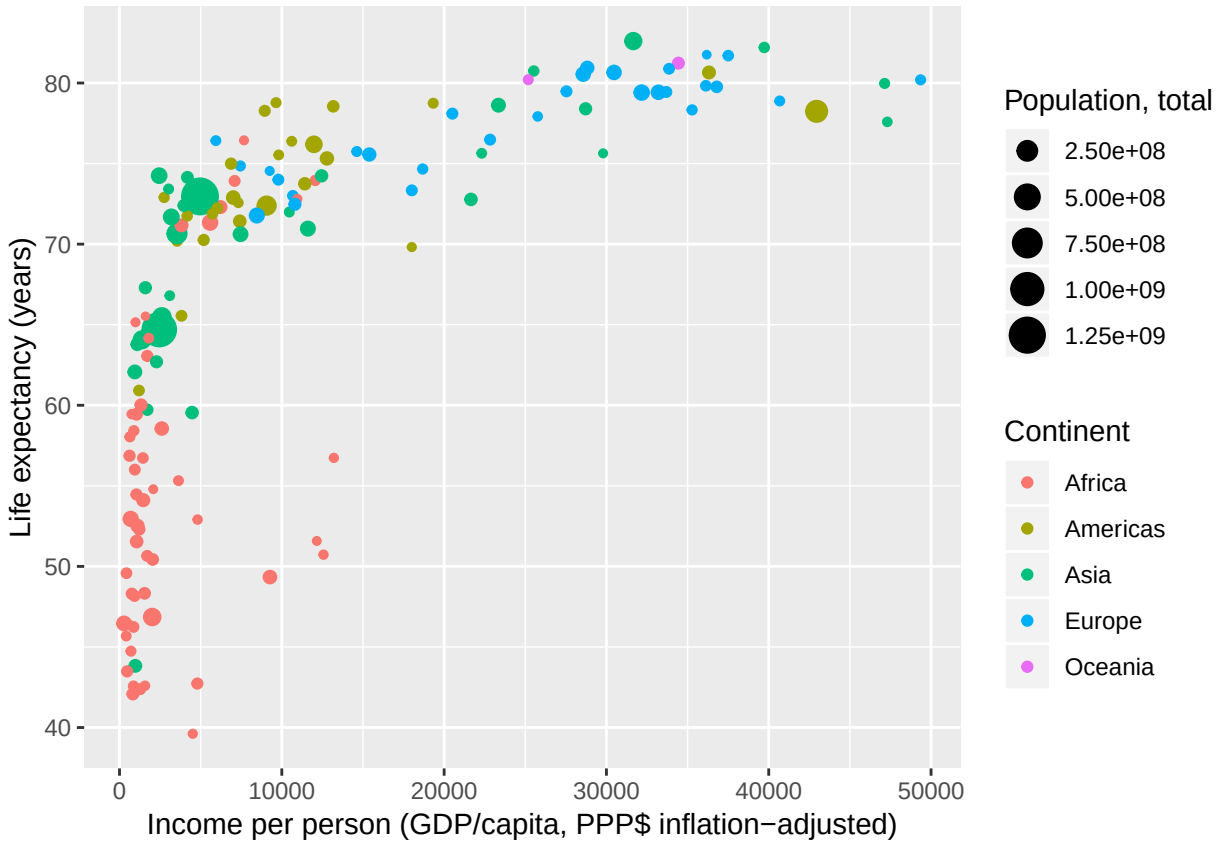
**When changing scale attributes we have to make sure to make the changes on the appropriate scale**. Just ask yourself:

1) What aesthetic does the scale correspond to? `scale_<name of aesthetic>_*`
2) Am I dealing with a *discrete* or *continuous* variable? `scale_*_<continuous|discrete>`

#### 4.2.2.1 Naming scales

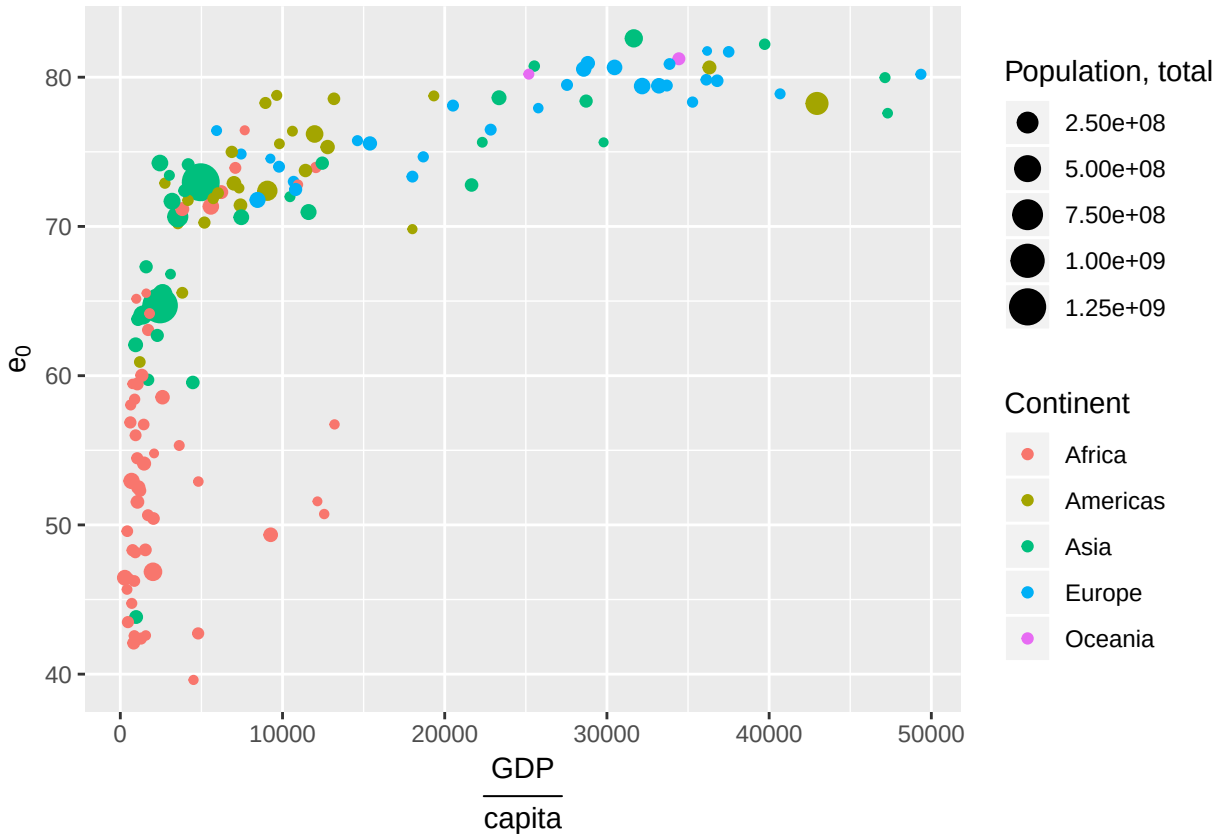
Once you know which scales to use, names are trivial to change.

```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = "Income per person (GDP/capita, PPP$ inflation-adjusted)") +
  scale_y_continuous(name = "Life expectancy (years)") +
  scale_color_discrete(name = "Continent") +
  scale_size_continuous(name = "Population, total")
```




You can also use mathematical annotation in your scale names. For further information consult `?plotmath`.

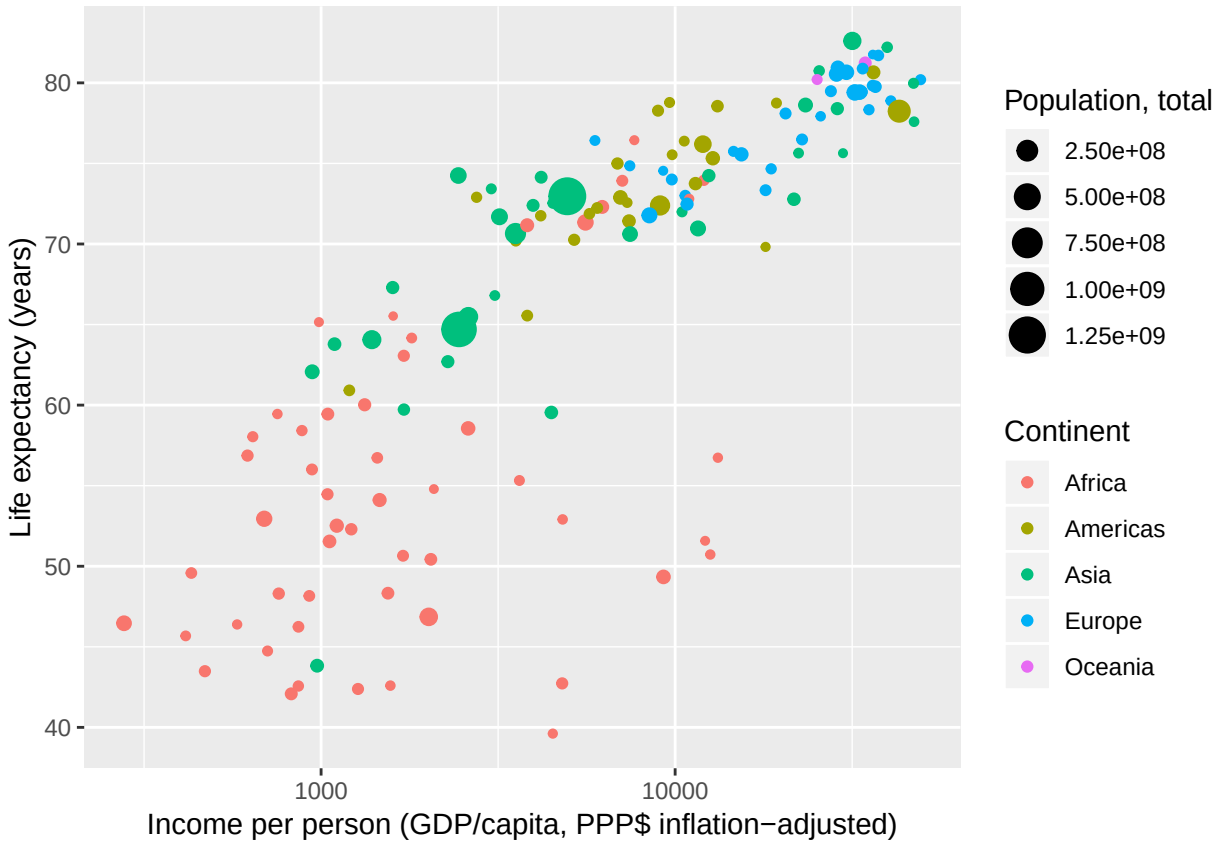
```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = expression(over(GDP, capita))) +
  scale_y_continuous(name = expression(e[0])) +
  scale_color_discrete(name = "Continent") +
  scale_size_continuous(name = "Population, total")
```




#### 4.2.2.2 Scale transformations

Next, we deal with *scale transformations*. In **Gapminder World** the x-axis is log-scaled meaning that the log of the x-axis data is taken before plotting. However, the labels remain on the linear scale. In that regard transforming scales is different from directly transforming the underlying data.

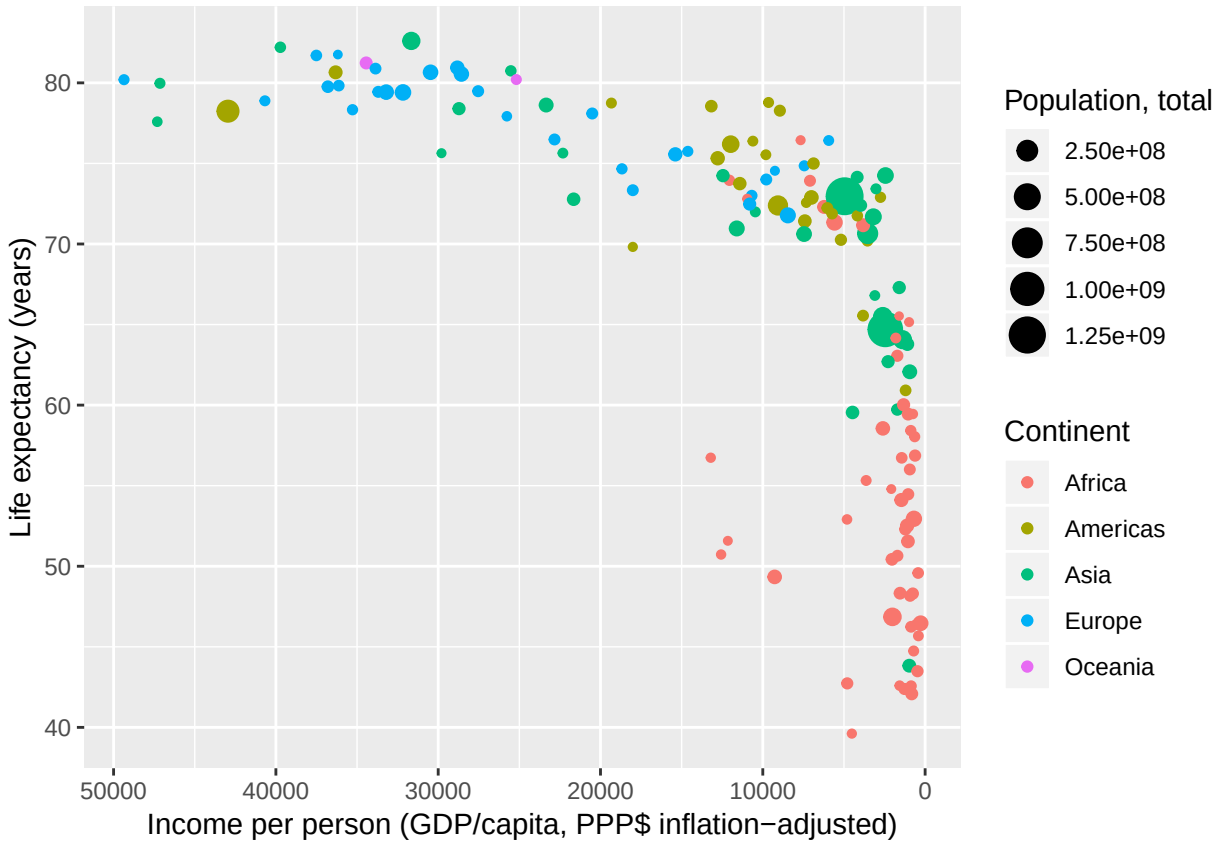
```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = "Income per person (GDP/capita, PPP$ inflation-adjusted)",
                     trans = "log10") +
  scale_y_continuous(name = "Life expectancy (years)") +
  scale_color_discrete(name = "Continent") +
  scale_size_continuous(name = "Population, total")
```




There are many different scale transformations built into ggplot. From the documentation:

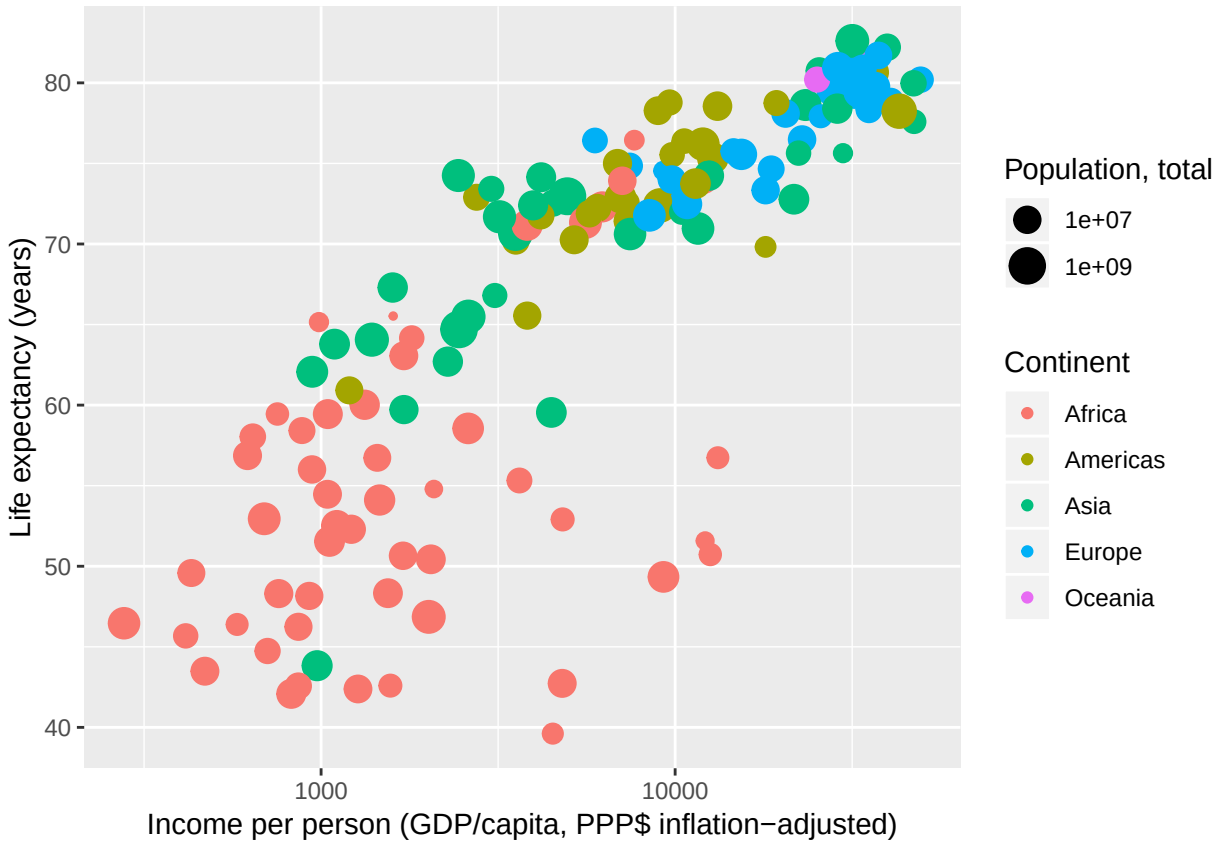
> Built-in transformations include "asn", "atanh", "boxcox", "exp", "identity", "log", "log10", "log1p", "log2", "logit", "probability", "probit", "reciprocal", "reverse" and "sqrt".

```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = "Income per person (GDP/capita, PPP$ inflation-adjusted)",
                     trans = "reverse") +
  scale_y_continuous(name = "Life expectancy (years)") +
  scale_color_discrete(name = "Continent") +
  scale_size_continuous(name = "Population, total")
```




Note that the concept of scale transformations is not limited to position scales.

```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = "Income per person (GDP/capita, PPP$ inflation-adjusted)",
                     trans = "log10") +
  scale_y_continuous(name = "Life expectancy (years)") +
  scale_color_discrete(name = "Continent") +
  scale_size_continuous(name = "Population, total", trans = "log10")
```

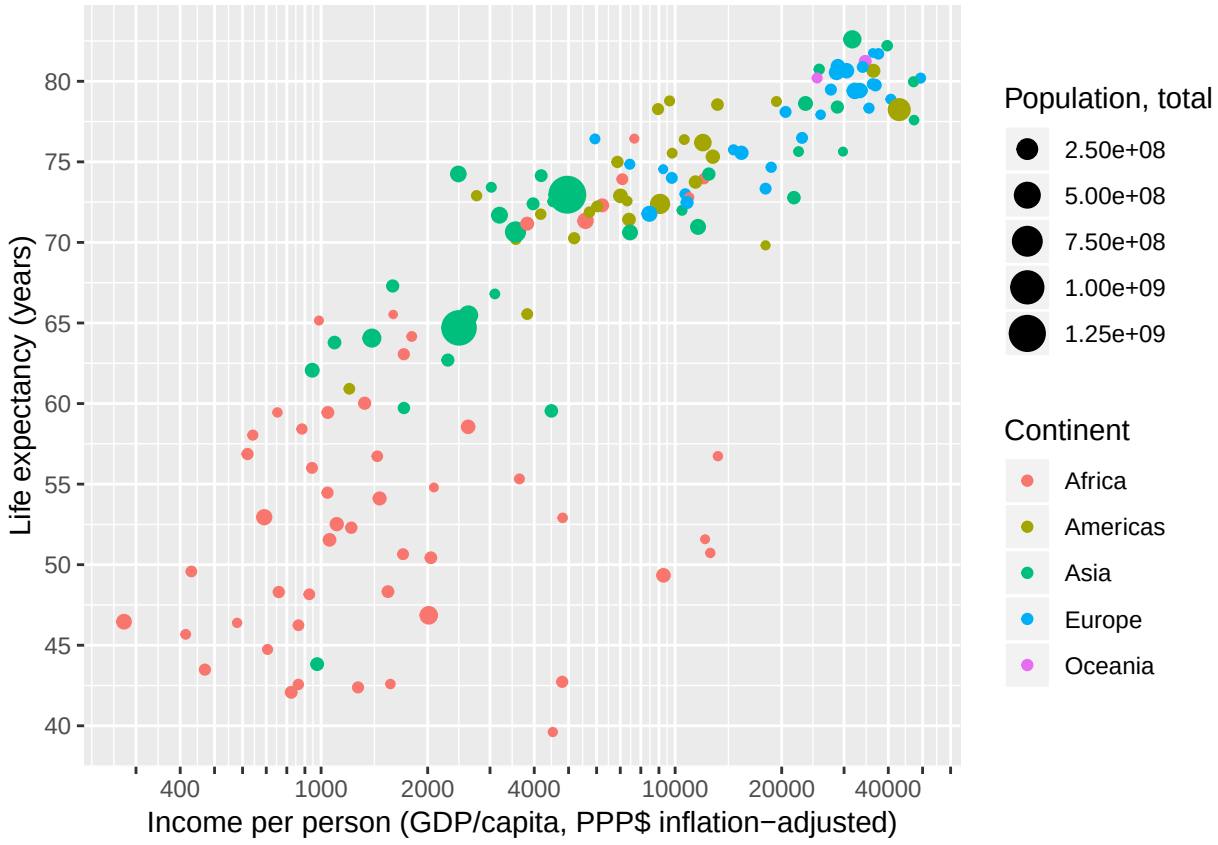



#### 4.2.2.3 Scale breaks and labels

Next, we manually specify the axis *breaks* and *labels* to be the same as in *Gapminder World*. Axis breaks are the positions where tick-marks and grid-lines are drawn. Labels specify what text to put at the breaks. **Breaks and labels have to be vectors of equal length.**

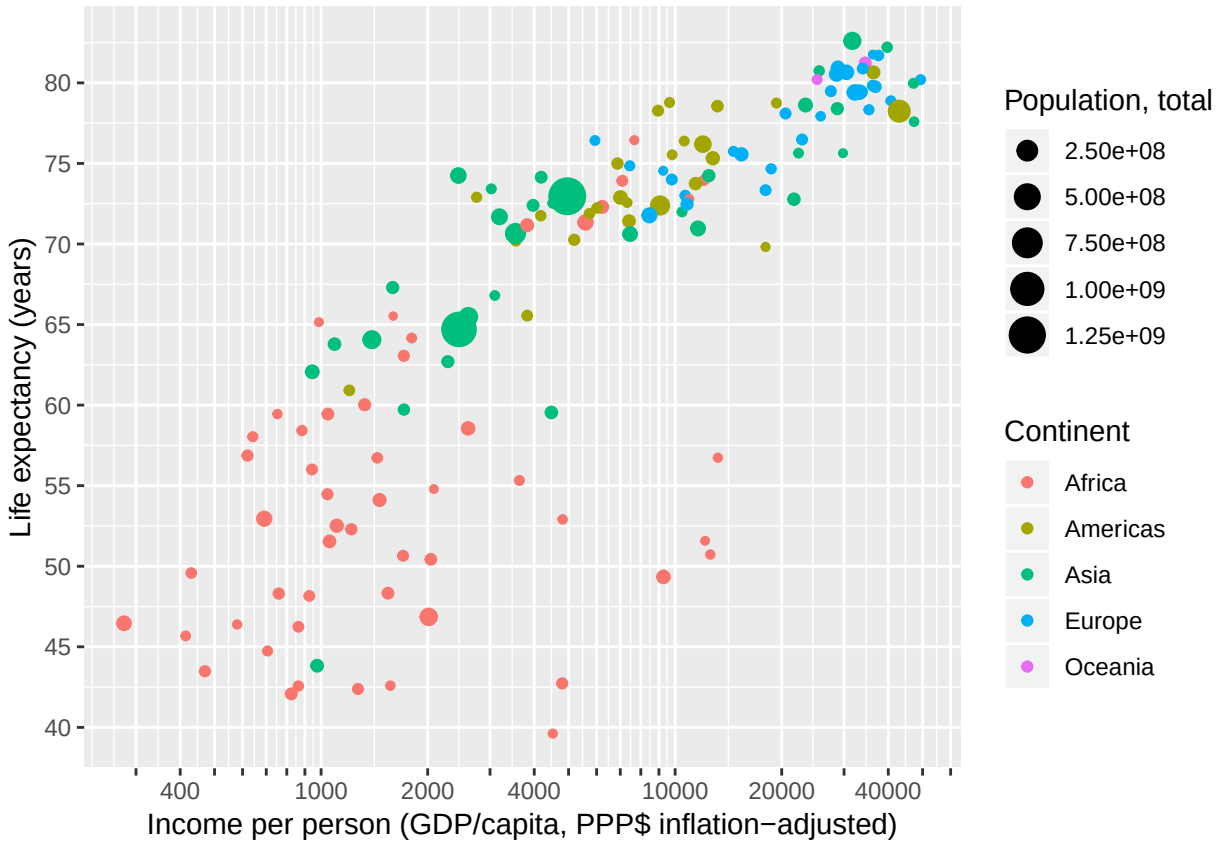
```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = "Income per person (GDP/capita, PPP$ inflation-adjusted)",
                     trans = "log10",
                     breaks = c(200, 300, 400, 500,
                                600, 700, 800, 900,
                                1000, 2000, 3000, 4000, 5000,
                                6000, 7000, 8000, 9000,
                                10000, 20000, 30000, 40000, 50000,
                                60000, 70000, 80000, 90000),
                     labels = c("200", "", "400", "",
                                "", "", "", "",
                                "1000", "2000", "", "4000", "",
                                "", "", "", "",
                                "10000", "20000", "", "40000", "",
                                "", "", "", "")) +
  scale_y_continuous(name = "Life expectancy (years)",
                     breaks = c(25, 30, 35, 40, 45, 50,
                                55, 60, 65, 70, 75, 80, 85)) +
```

```
  scale_color_discrete(name  = "Continent") +
  scale_size_continuous(name = "Population, total")
```



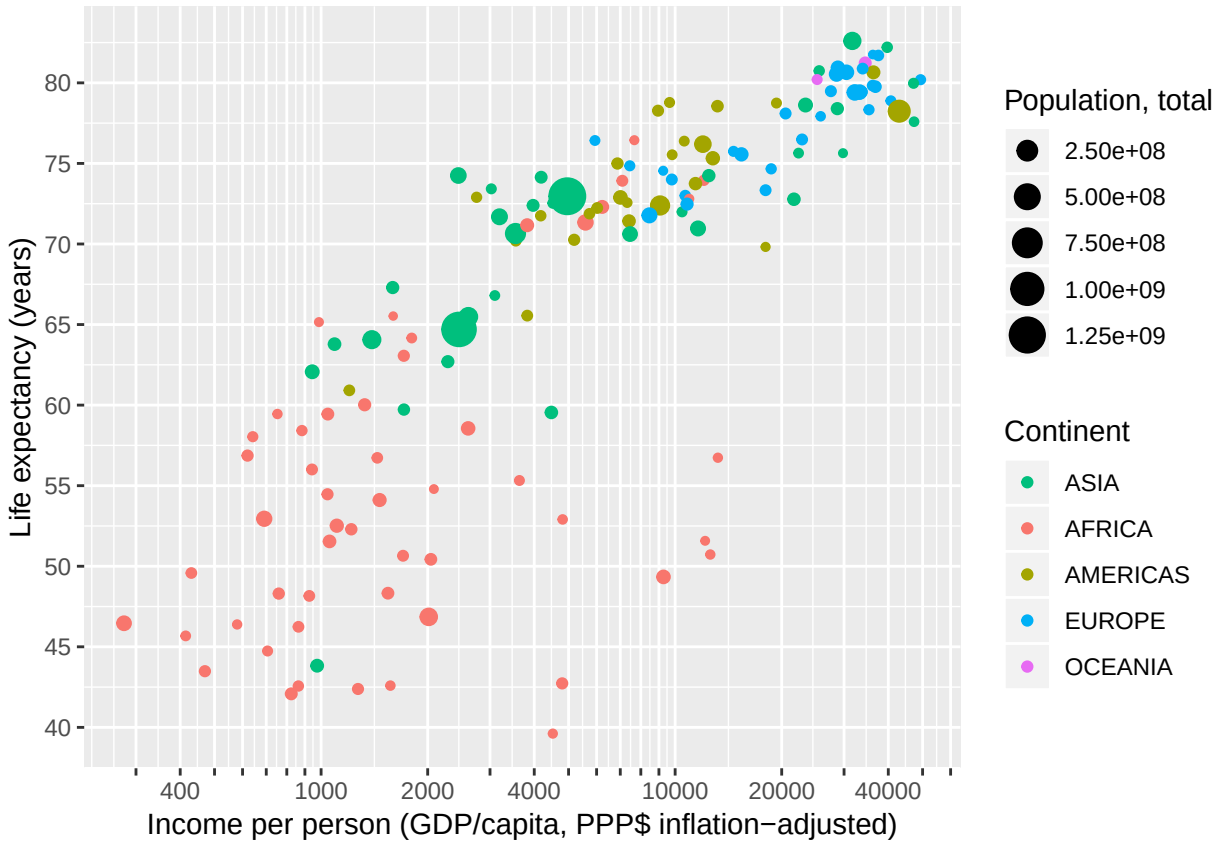

OK, that was effective but clumsy. Luckily ggplot does not care *how* we generate the vector of breaks. We can use any R function as long as it outputs a vector. Even better, instead of manually spelling out the labels for each break we can write a short function that takes the breaks as input and formats them. Much nicer code – same result.

```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = "Income per person (GDP/capita, PPP$ inflation-adjusted)",
                     trans = "log10",
                     breaks = apply(expand.grid(1:9, 10^(2:4)), 1, FUN = prod)[-1],
                     labels = function(x) ifelse(grepl('^[124]', x), x, '')) +
  scale_y_continuous(name = "Life expectancy (years)",
                     breaks = seq(25, 85, 5)) +
  scale_color_discrete(name = "Continent") +
  scale_size_continuous(name = "Population, total")
```



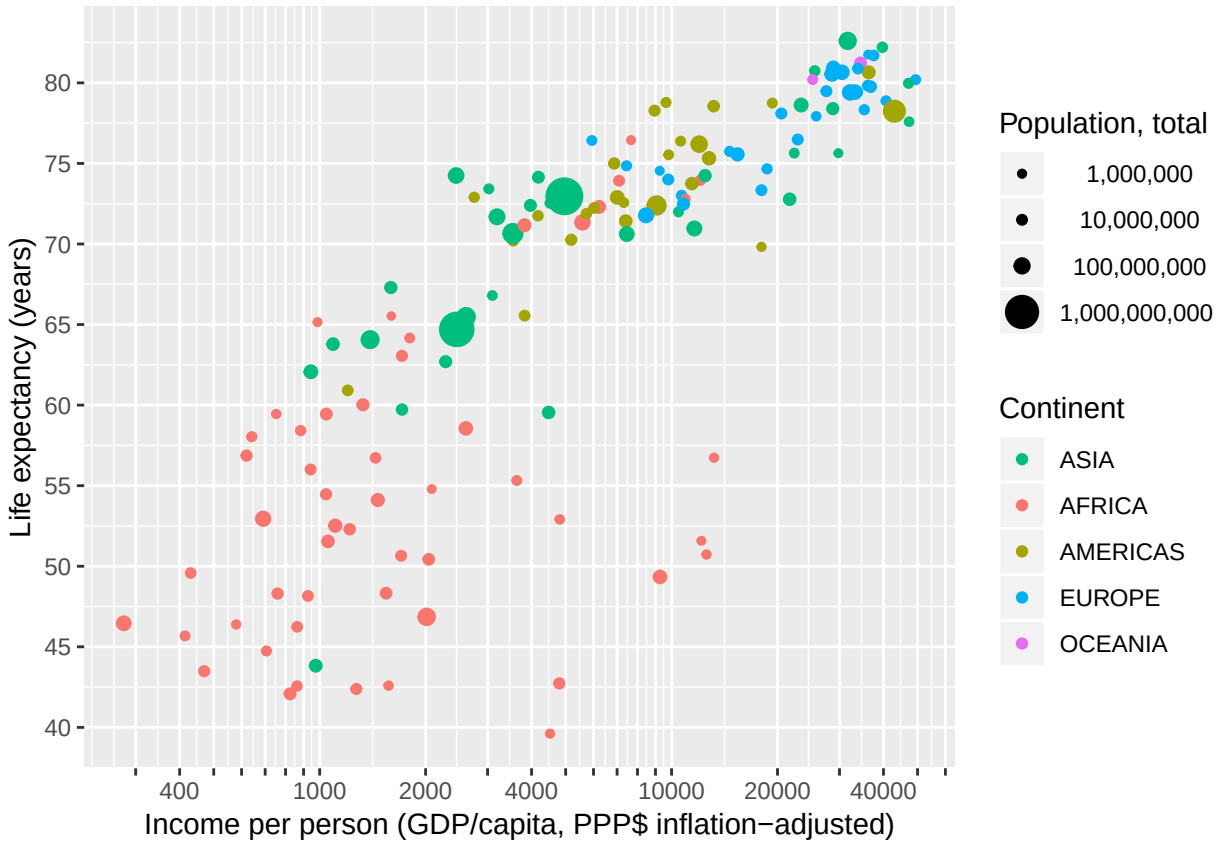

The concept of *breaks* and *labels* does not only apply to continuous axis. **All scales have breaks and labels**. E.g. on a colour scale the breaks are the colour keys, the labels are – well – the labels. **We reorder the items on our discrete scale by specifying the breaks in the required order.** We also use an R function to capitalize the labels.

```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = "Income per person (GDP/capita, PPP$ inflation-adjusted)",
                     trans = "log10",
                     breaks = apply(expand.grid(1:9, 10^(2:4)), 1, FUN = prod)[-1],
                     labels = function(x) ifelse(grepl("^[124]", x), x, "")) +
  scale_y_continuous(name = "Life expectancy (years)",
                     breaks = seq(25, 85, 5)) +
  scale_color_discrete(name = "Continent",
                       breaks = c("Asia", "Africa", "Americas", "Europe", "Oceania"),
                       labels = toupper) +
  scale_size_continuous(name = "Population, total")
```




Finally, let's choose some sensible breaks and labels for the size scale.

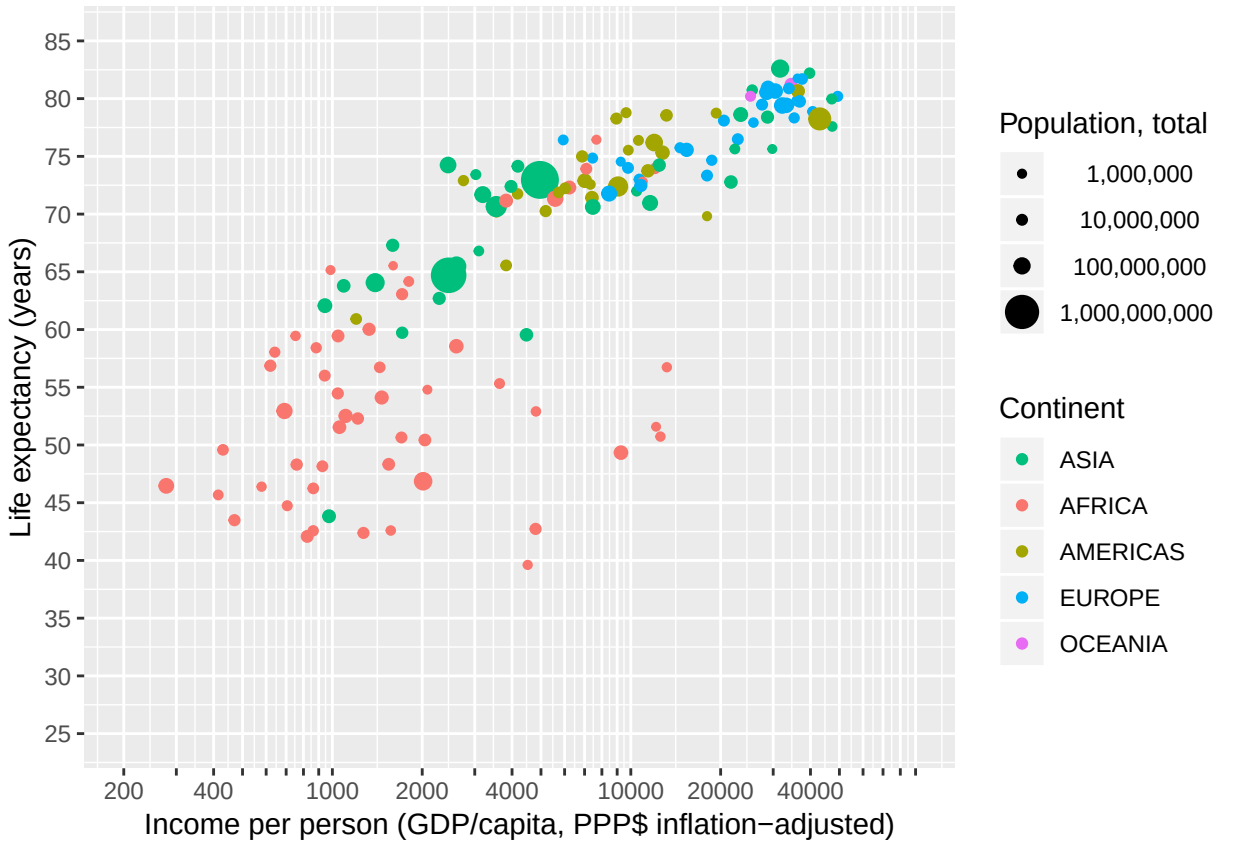
```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = "Income per person (GDP/capita, PPP$ inflation-adjusted)",
                     trans = "log10",
                     breaks = apply(expand.grid(1:9, 10^(2:4)), 1, FUN = prod)[-1],
                     labels = function(x) ifelse(grepl("^[124]", x), x, "")) +
  scale_y_continuous(name = "Life expectancy (years)",
                     breaks = seq(25, 85, 5)) +
  scale_color_discrete(name = "Continent",
                       breaks = c("Asia", "Africa", "Americas", "Europe", "Oceania"),
                       labels = toupper) +
  scale_size_continuous(name = "Population, total",
                        breaks = c(1E6, 10E6, 100E6, 1E9),
                        labels = function(x) format(x, big.mark = ",", scientific = FALSE))
```




#### 4.2.2.4 Scale limits

We match the maximum and minimum value of our xy-scales with those of the *Gapminder World* chart by specifying the *limits* of the scales.

```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop, colour = continent)) +
  geom_point() +
  scale_x_continuous(name = "Income per person (GDP/capita, PPP$ inflation-adjusted)",
                     trans = "log10",
                     breaks = apply(expand.grid(1:9, 10^(2:4)), 1, FUN = prod)[-1],
                     labels = function(x) ifelse(grepl("^[124]", x), x, ""),
                     limits = c(200, 90000)) +
  scale_y_continuous(name = "Life expectancy (years)",
                     breaks = seq(25, 85, 5),
                     limits = c(25, 85)) +
  scale_color_discrete(name = "Continent",
                       breaks = c("Asia", "Africa", "Americas", "Europe", "Oceania"),
                       labels = toupper) +
  scale_size_continuous(name = "Population, total",
                        breaks = c(1E6, 10E6, 100E6, 1E9),
                        labels = function(x) format(x, big.mark = ",", scientific = FALSE))
```

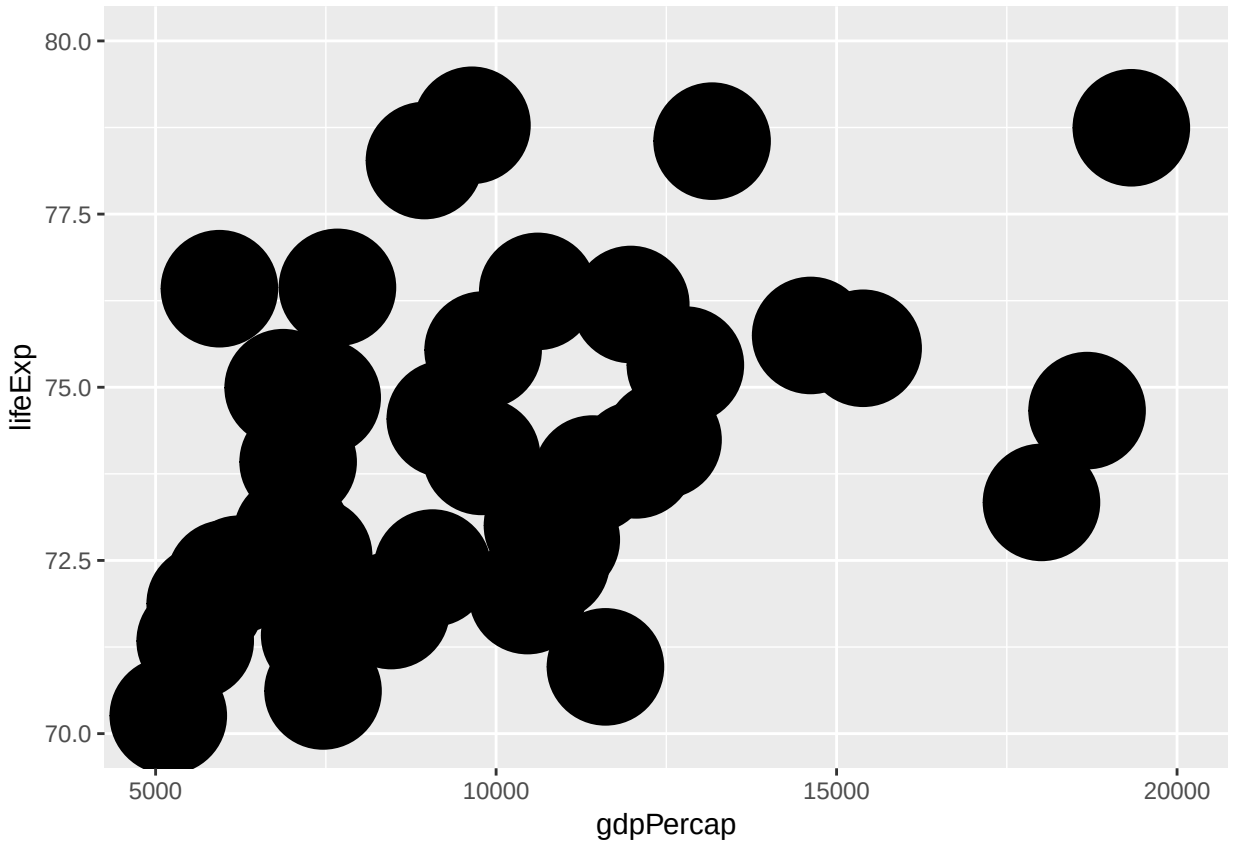



#### 4.2.2.5 Limiting versus zooming

Note that values outside of the limits will be discarded. This is of importance if you want to zoom into a plot. Here we "zoom" by changing the limits of the scales...

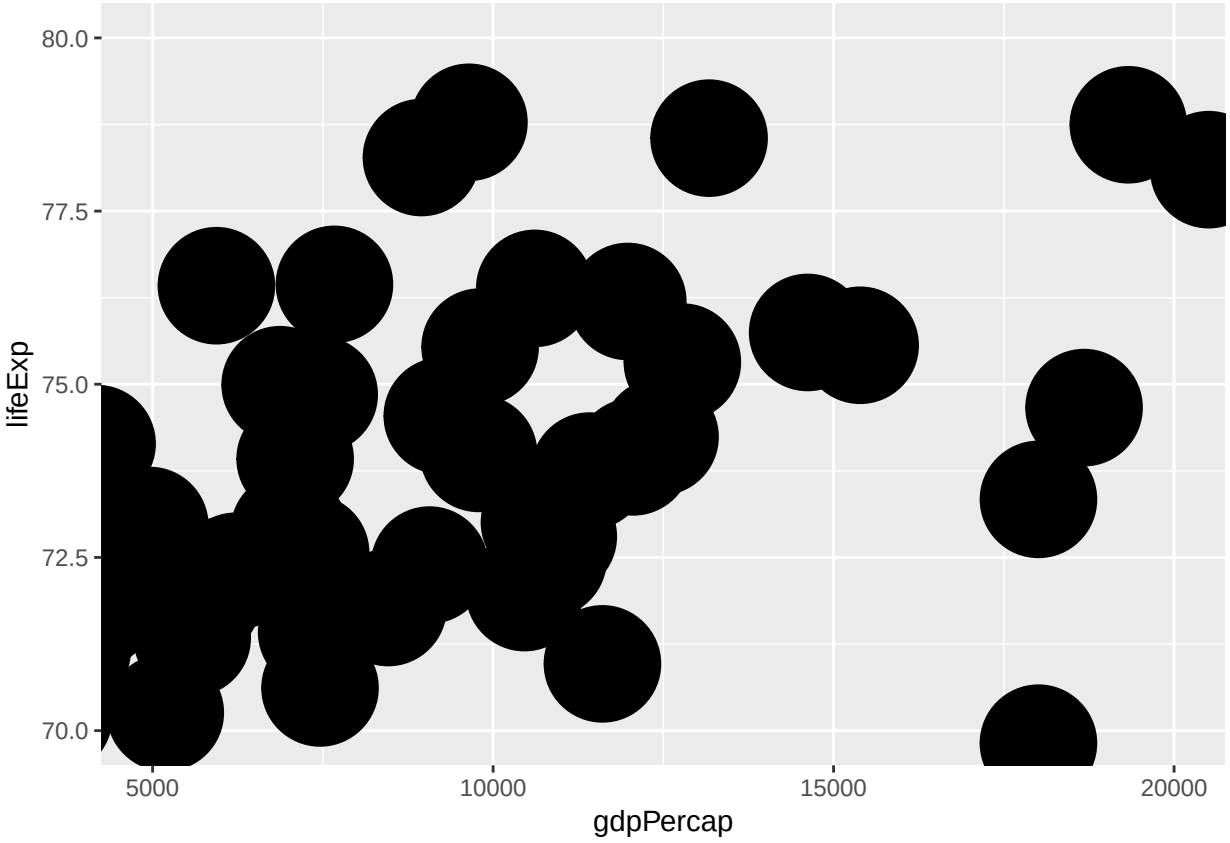
```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point(size = 20) +
  scale_x_continuous(limits = c(5000, 20000)) +
  scale_y_continuous(limits = c(70, 80))
```

```
## Warning: Removed 104 rows containing missing values (geom_point).
```




...and here we zoom by changing the limits of the coordinate system

```
gapminder %>% filter(year == 2007) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point(size = 20) +
  coord_cartesian(xlim = c(5000, 20000), ylim = c(70, 80))
```




### 4.2.3 Excercise: The grammar of graphics

## 4.3 Order

### 4.3.1 Factor levels control the order of discrete scales

It is easy to order items on a numerical scale. One just puts them on the number line. Usually low on the left and hight to the right. But what about discrete items? ggplot orders them according to the order of their factor levels. An example:

```
# test data
foo <- data.frame(id  = 1:4,
                  sex = c("Female", "Female", "Male", "Male"))
foo
```

```
##   id    sex
## 1  1 Female
## 2  2 Female
## 3  3   Male
## 4  4   Male
```

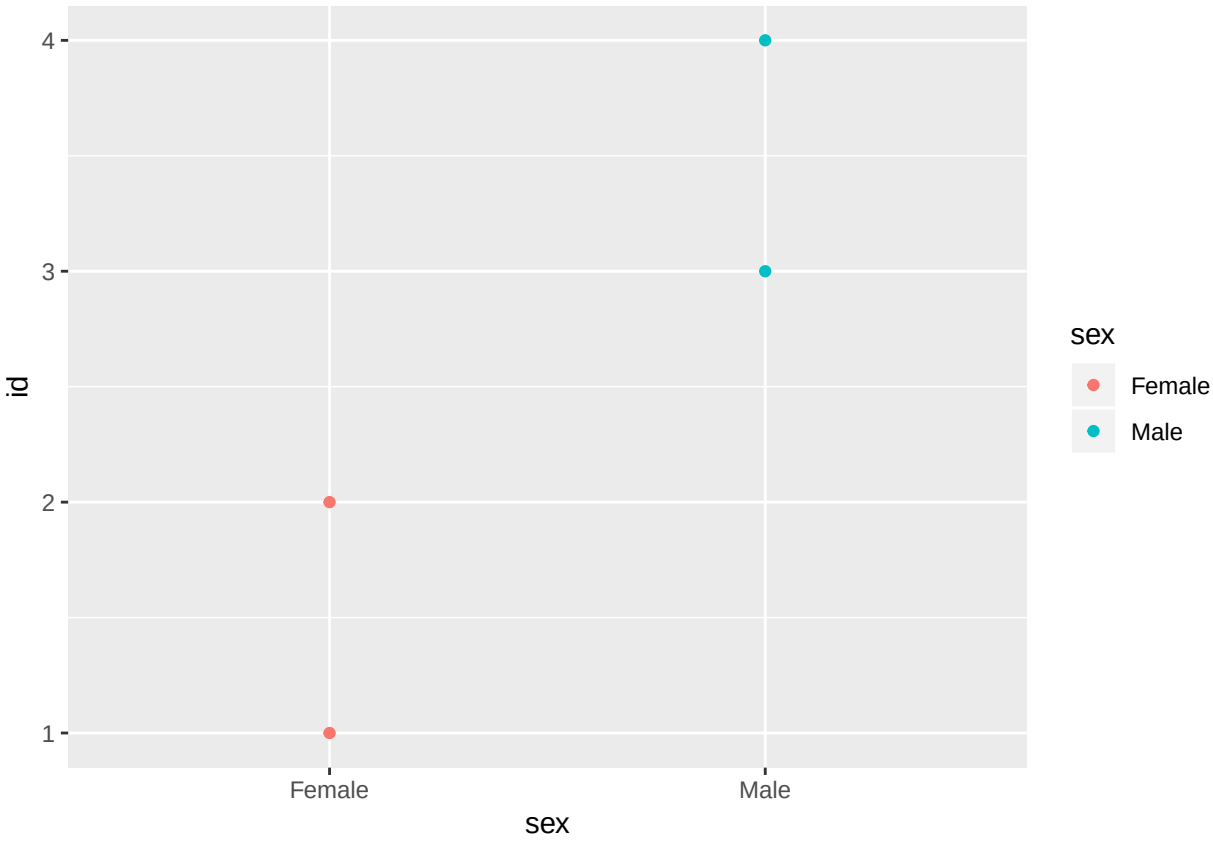
`data.frame`, just like ggplot, automatically converts a character vector to a factor using `as.factor`. The levels order of that factor follows the sequence of occurrence in the data.

```
levels(foo$sex)
```

```
## [1] "Female" "Male"
```

ggplot constructs discrete scales in the order of the levels of the underlying factor variable. Here, Females first, males after.

```
ggplot(foo) +
  geom_point(aes(x = sex, y = id, color = sex))
```




If we reverse the level order of the sex variable we change the way ggplot orders the discrete items.

```
foo$sex
```

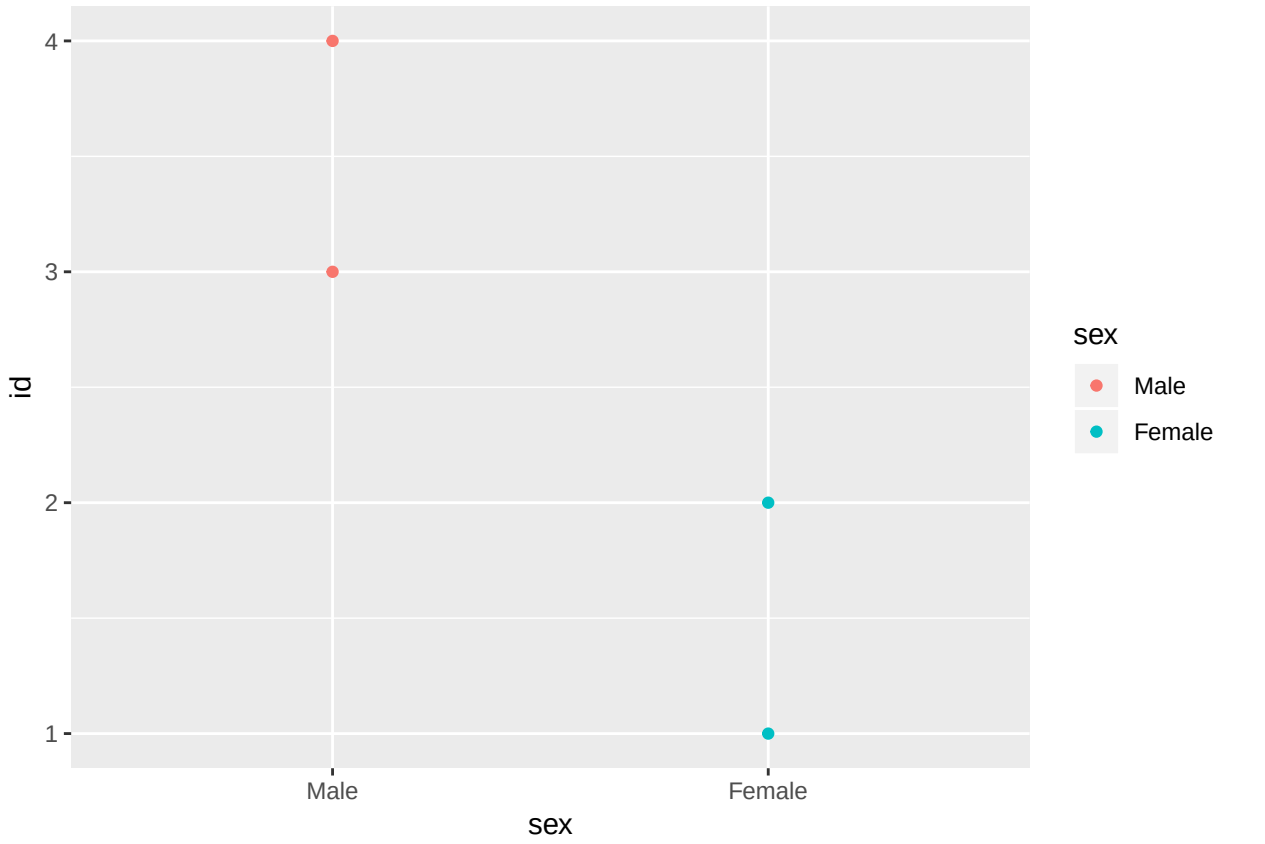
```
## [1] Female Female Male   Male
## Levels: Female Male
```

```
foo$sex <- factor(foo$sex, levels = c("Male", "Female"))
foo$sex
```

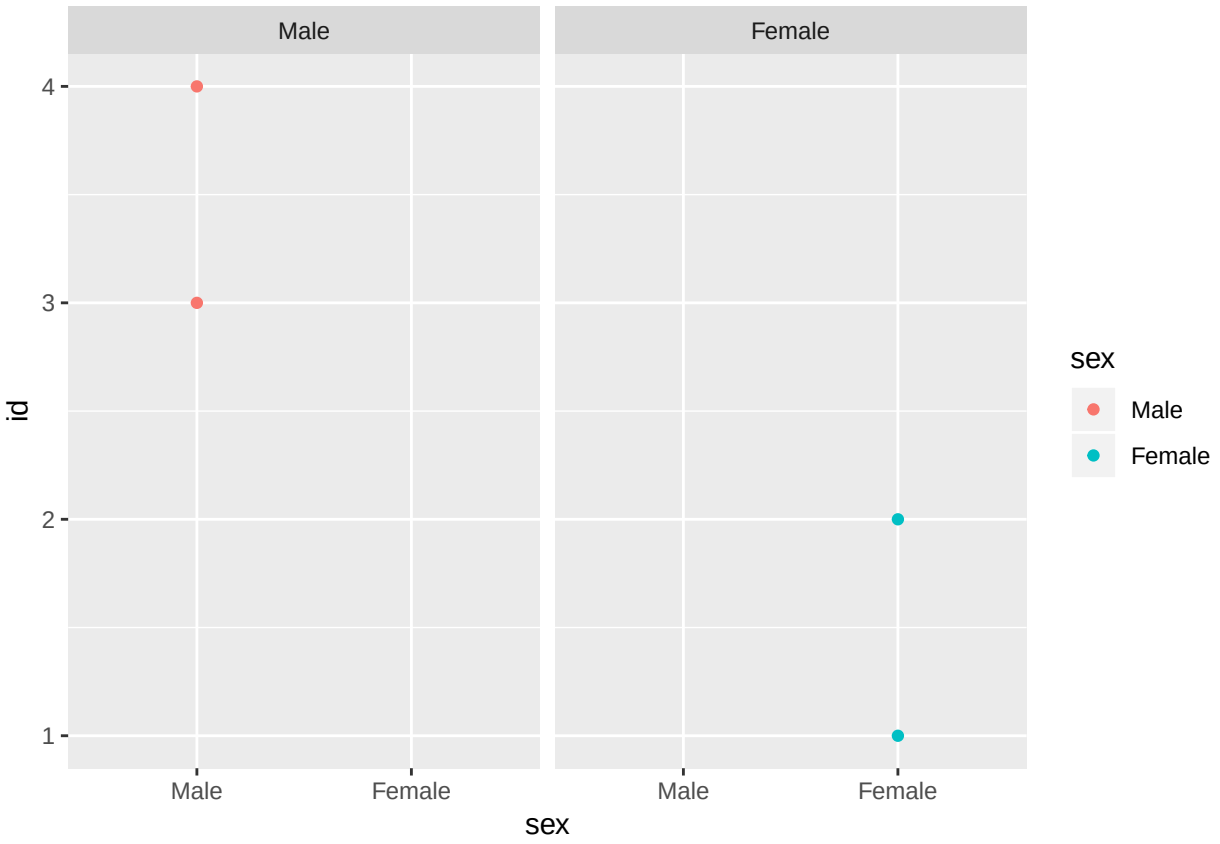
```
## [1] Female Female Male   Male
## Levels: Male Female
```

Now we have males first and females last.

```
ggplot(foo) +
  geom_point(aes(x = sex, y = id, color = sex))
```




```
ggplot(foo) +
  geom_point(aes(x = sex, y = id, color = sex)) +
  facet_wrap(~sex)
```




**NEVER OVERRIDE THE LEVELS DIRECTLY WHEN JUST MEANING TO CHANGE THE ORDER!** You'll screw up your data. In our case we just changed the sex of the participants.

```
foo$sex
```

```
## [1] Female Female Male   Male
## Levels: Male Female
```

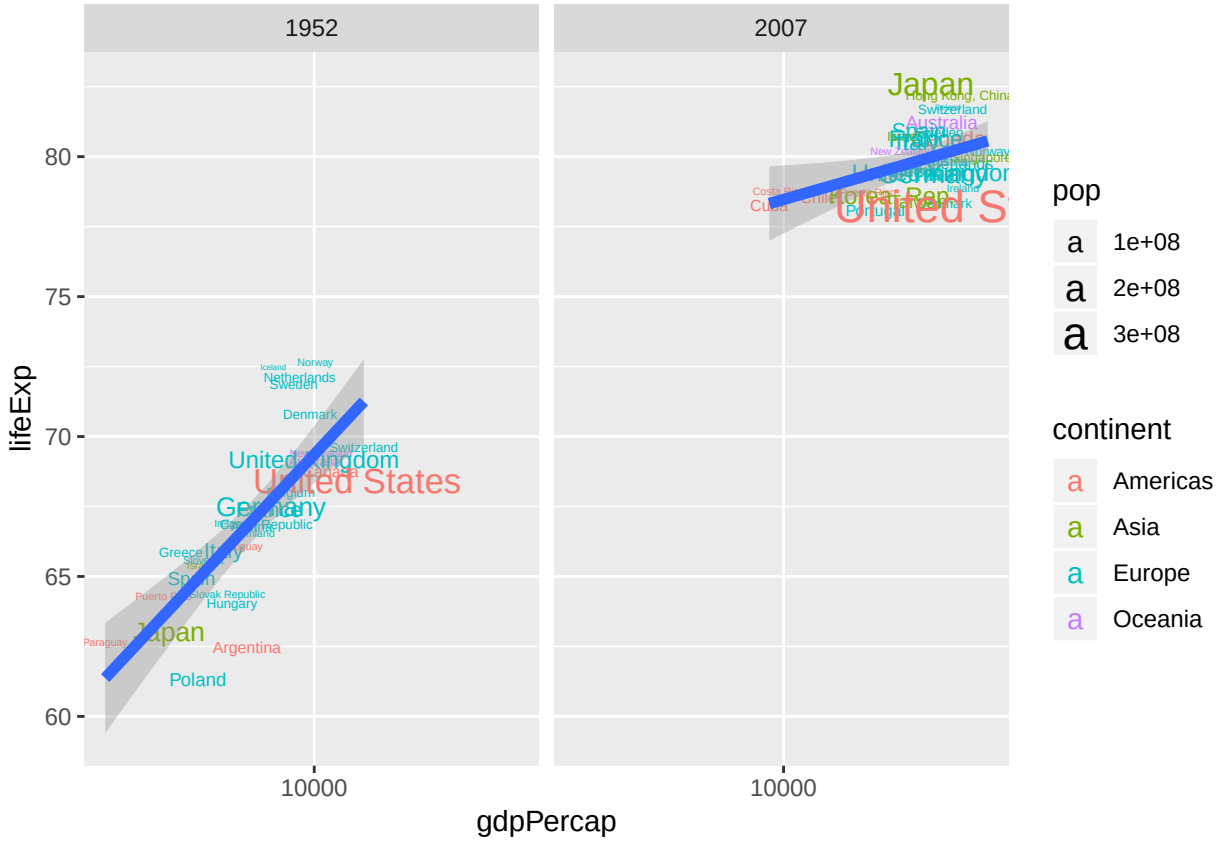
```
levels(foo$sex) <- c("Female", "Male")
foo$sex
```

```
## [1] Male   Male   Female Female
## Levels: Female Male
```

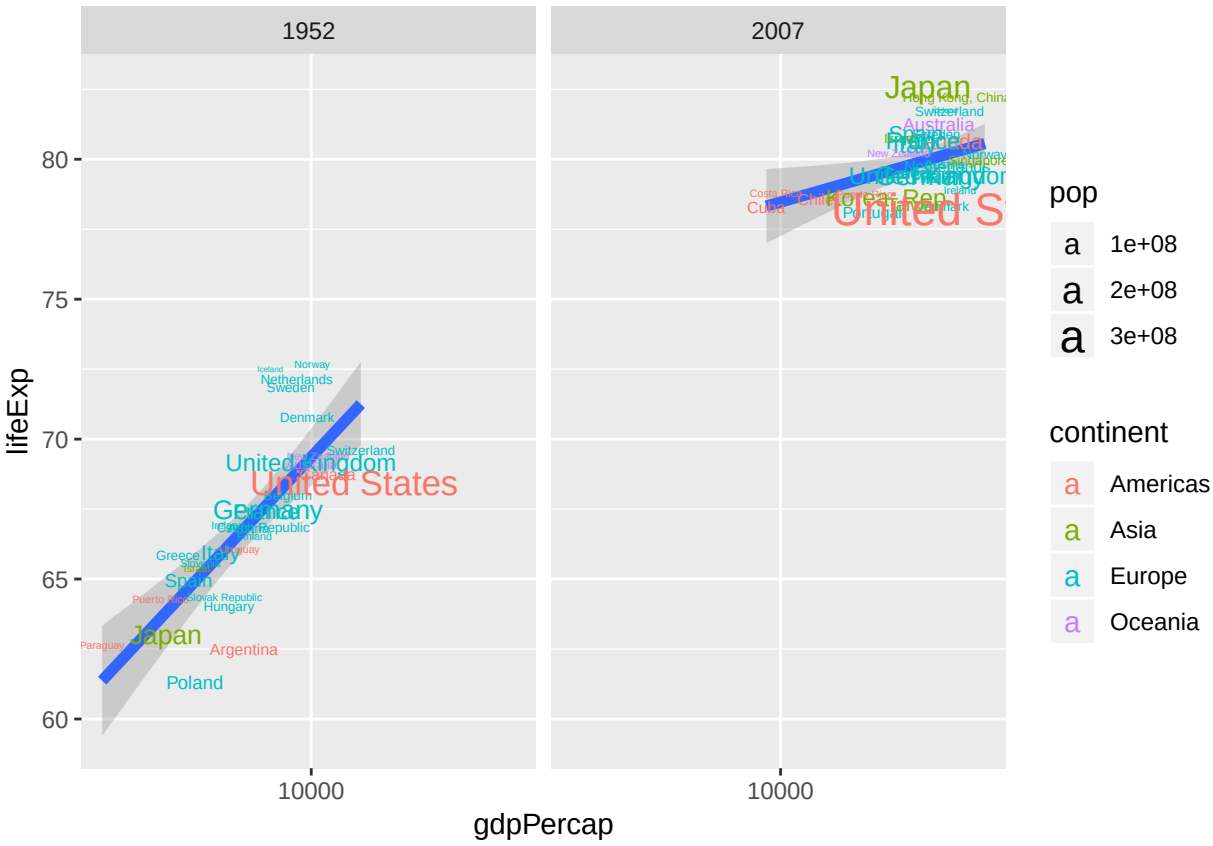
### 4.3.2 Layers are plotted on top of each other

```
library(gapminder)

gapminder %>%
  filter(year %in% c(1952, 2007)) %>%
  group_by(year) %>%
  mutate(e0rank = n() - rank(lifeExp)) %>%
  filter(e0rank <= 30) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_text(aes(label = country, color = continent, size = pop)) +
  scale_x_log10() +
  geom_smooth(method = "lm", size = 2) +
```

```
  facet_wrap(~year)
```



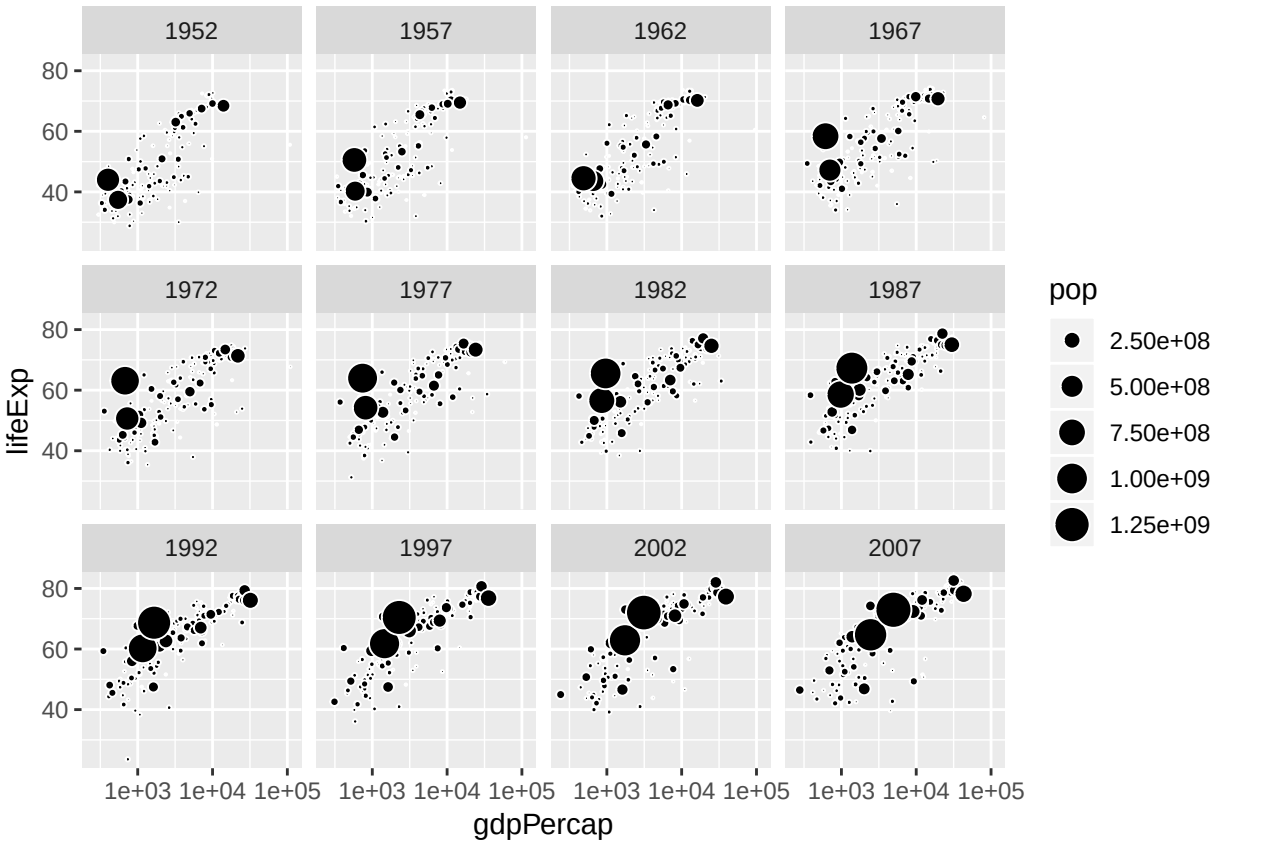

```
gapminder %>%
  filter(year %in% c(1952, 2007)) %>%
  group_by(year) %>%
  mutate(e0rank = n() - rank(lifeExp)) %>%
  filter(e0rank <= 30) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_smooth(method = "lm", size = 2) +
  geom_text(aes(label = country, color = continent, size = pop)) +
  scale_x_log10() +
  facet_wrap(~year)
```



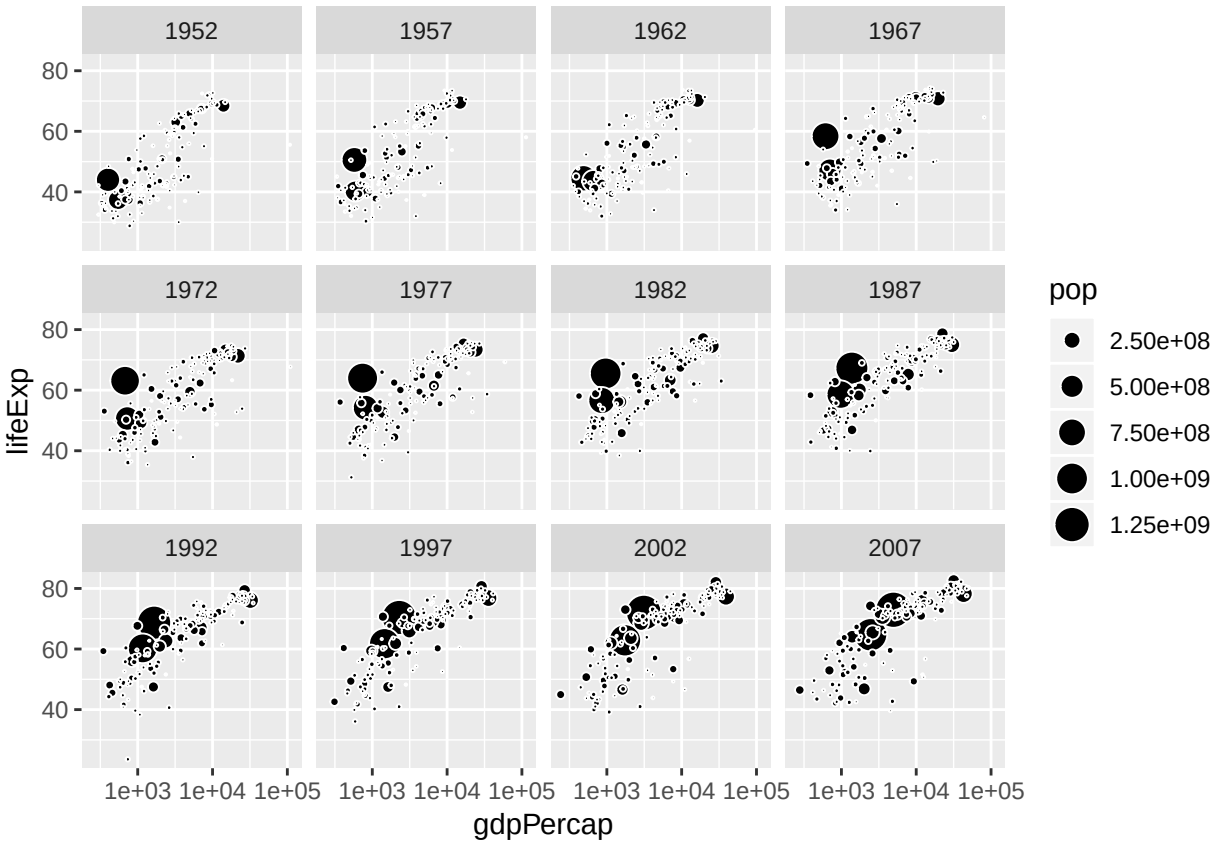

### 4.3.3 Sometimes the row-order of the data matters

Sometimes the row-order of a dataframe has implications for visualization. This is true for ggplots `geom_point()` which draws points in the order of occurence in a dataframe. In the example below we scale the points by the population size of a country. Arranging the dataframe by decreasing population size makes sure that the smaller points are drawn last, i.e. on top of the larger points.

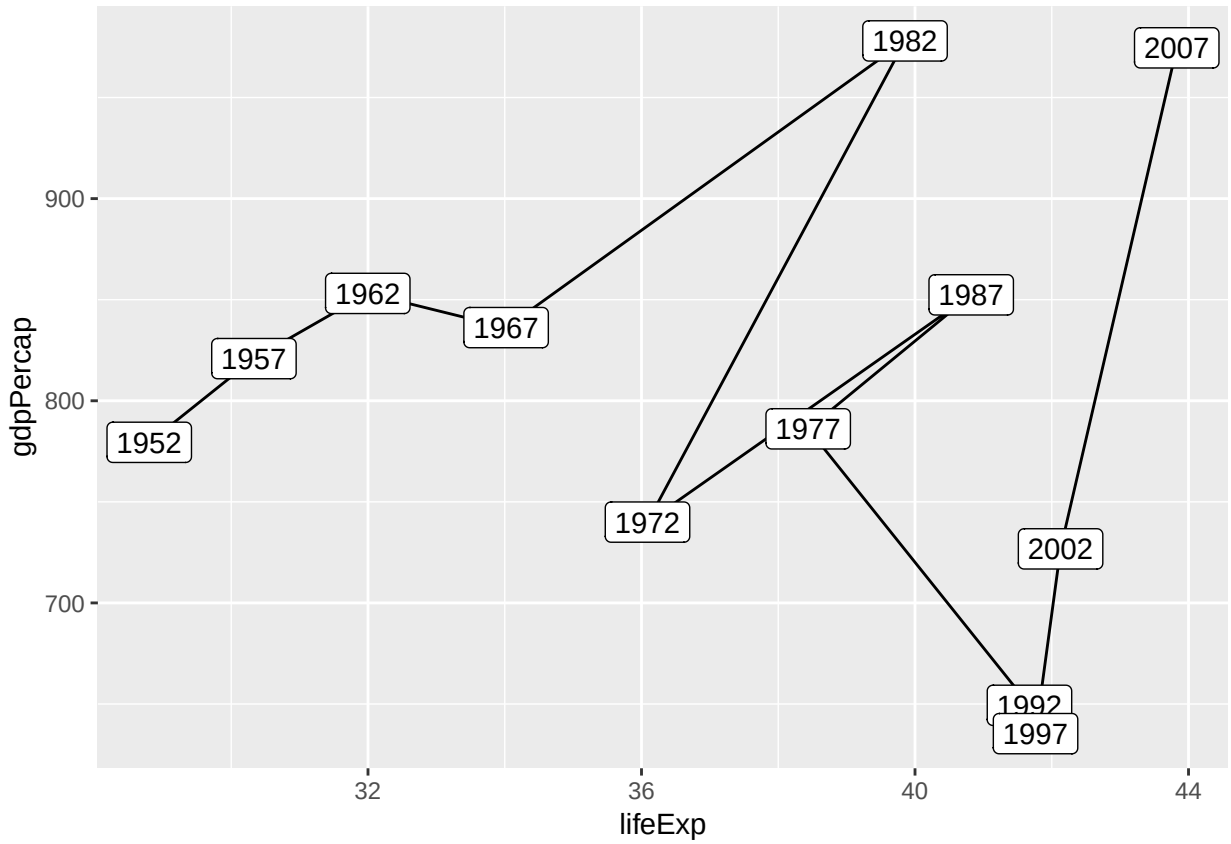
```
gapminder %>%
  arrange(pop) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop)) +
  geom_point(fill = 'black', color = 'white', shape = 21) +
  scale_x_log10() +
  scale_size_area() +
  facet_wrap(~year)
```



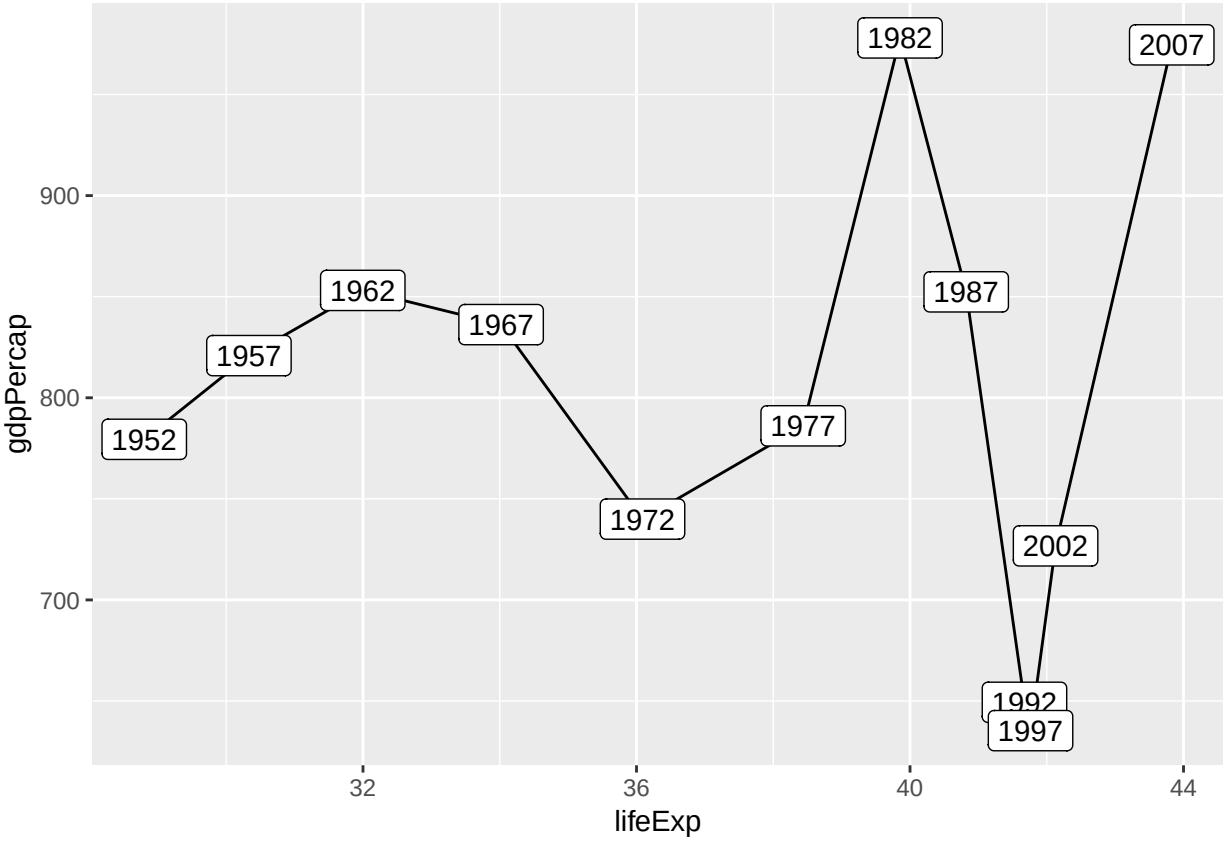

```
gapminder %>%
  arrange(desc(pop)) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, size = pop)) +
  geom_point(fill = 'black', color = 'white', shape = 21) +
  scale_x_log10() +
  scale_size_area() +
  facet_wrap(~year)
```




```
gapminder %>%
  filter(country == 'Afghanistan') %>%
  arrange(pop) %>%
  ggplot(aes(x = lifeExp, y = gdpPercap)) +
  geom_path() +
  geom_label(aes(label = year))
```




```
gapminder %>%
  filter(country == 'Afghanistan') %>%
  arrange(year) %>%
  ggplot(aes(x = lifeExp, y = gdpPercap)) +
  geom_path() +
  geom_label(aes(label = year))
```




## 4.4 Facets

### 4.4.1 Comparing sub-groups to totals

https://drsimonj.svbtle.com/plotting-background-data-for-groups-with-ggplot2

## 4.5 Chart type vocabulary

### 4.5.1 Visualizing distributions

Distribiutions are at the heart of statistics as they inform us about about the "typical" in a population, the degree of deviation from the typical, the "extreme", and everything in between. Visualizing the distribution of a collection of values allows us to

- appreciate the diversity in our sample,
- understand the variability around a mean,
- choose the correct model specification,
- decide if the data is iid.

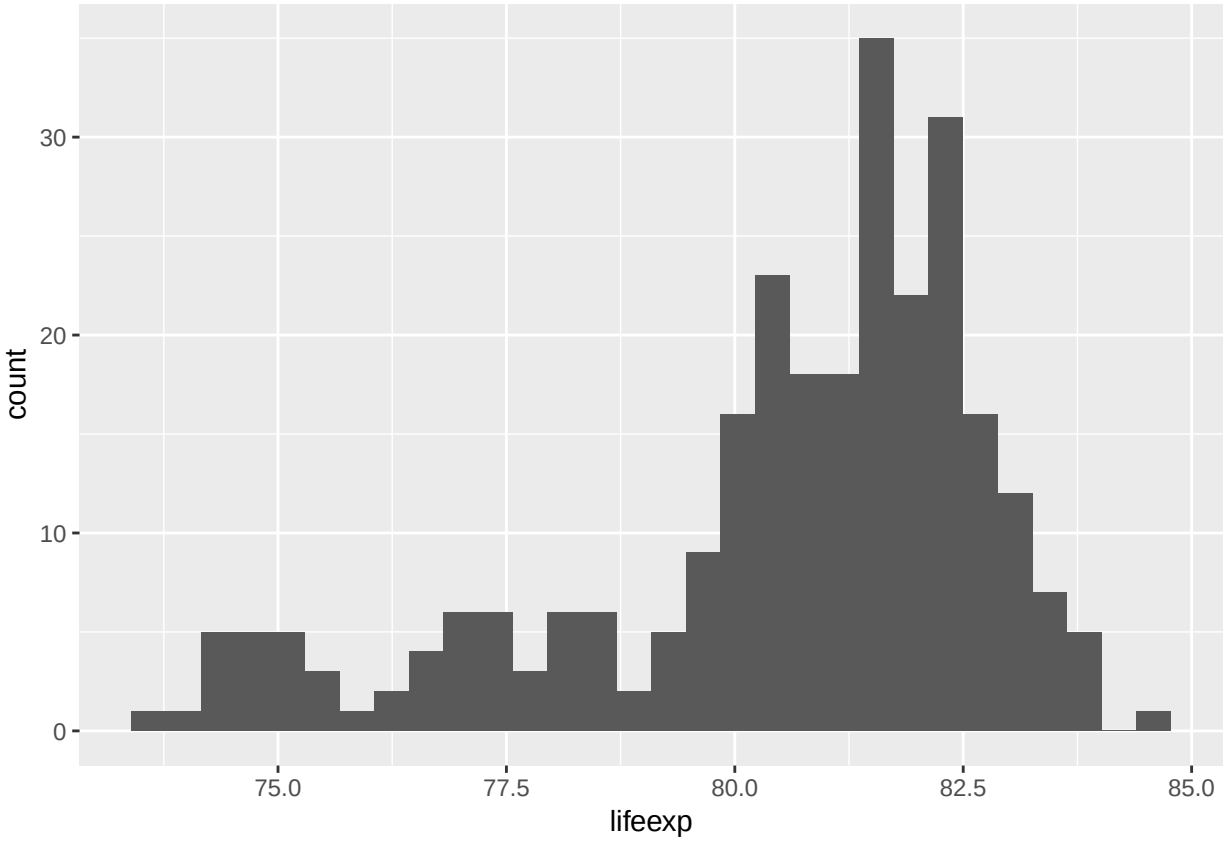
#### 4.5.1.1 Continuous

```
# European regional population statistics
load('data/euro_regio/euro_regio_eu.Rdata')
```

The most popular technique to visualize a distribution is the **histogram**. Here we use it to show how life-expectancy is distributed across the regions of the European Union in 2015.

```
euro_regio_eu %>%
  filter(year == 2015) %>%
  ggplot() +
  geom_histogram(aes(x = lifeexp))
```

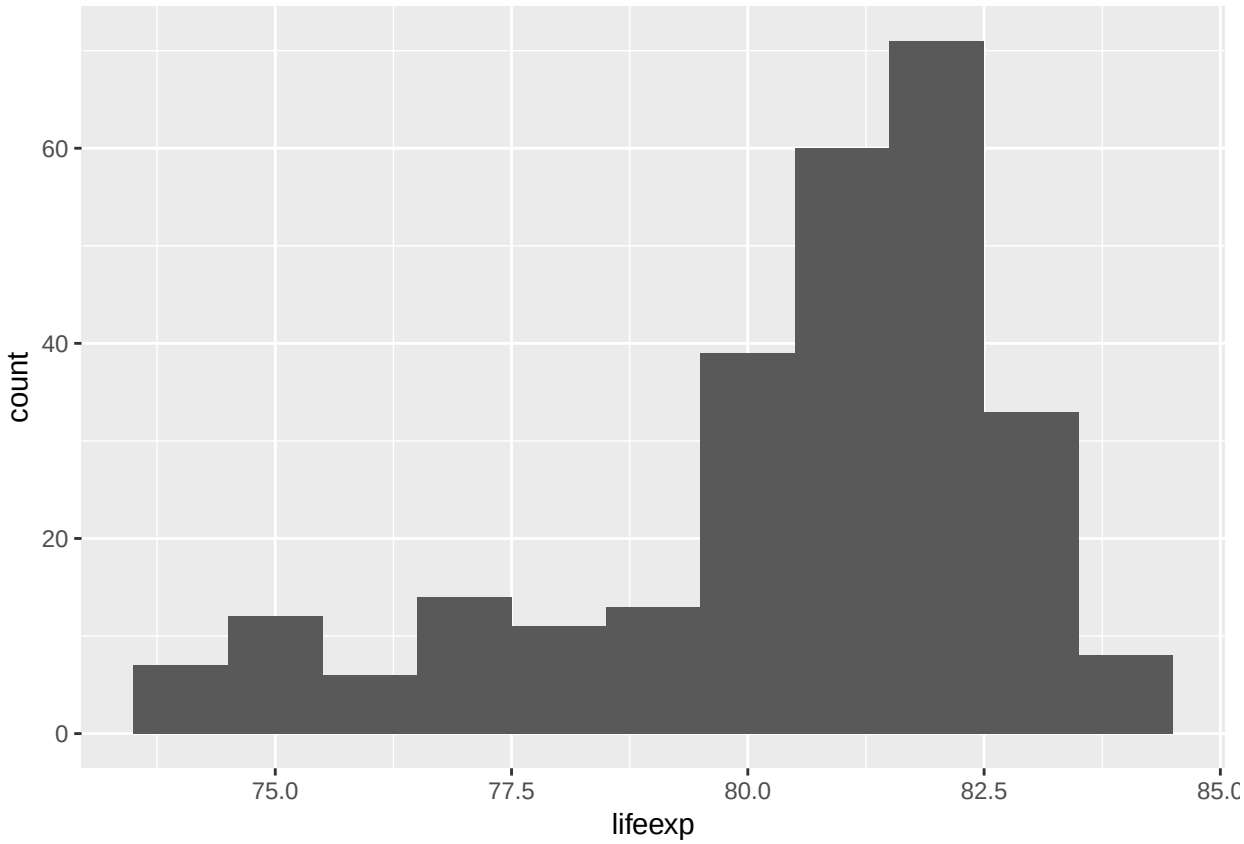
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.

## Warning: Removed 18 rows containing non-finite values (stat_bin).
```




A histogram is a visual representation of *binned data*: a variable gets cut into intervals, the cases tabulated over those intervals and the resulting counts plotted as a shaded region. This whole binning operation is performed by `geom_histogram()`. We can change the details of that data transformation, e.g. using single year bins.

```
euro_regio_eu %>%
  filter(year == 2015) %>%
  ggplot() +
  geom_histogram(aes(x = lifeexp), binwidth = 1)
```

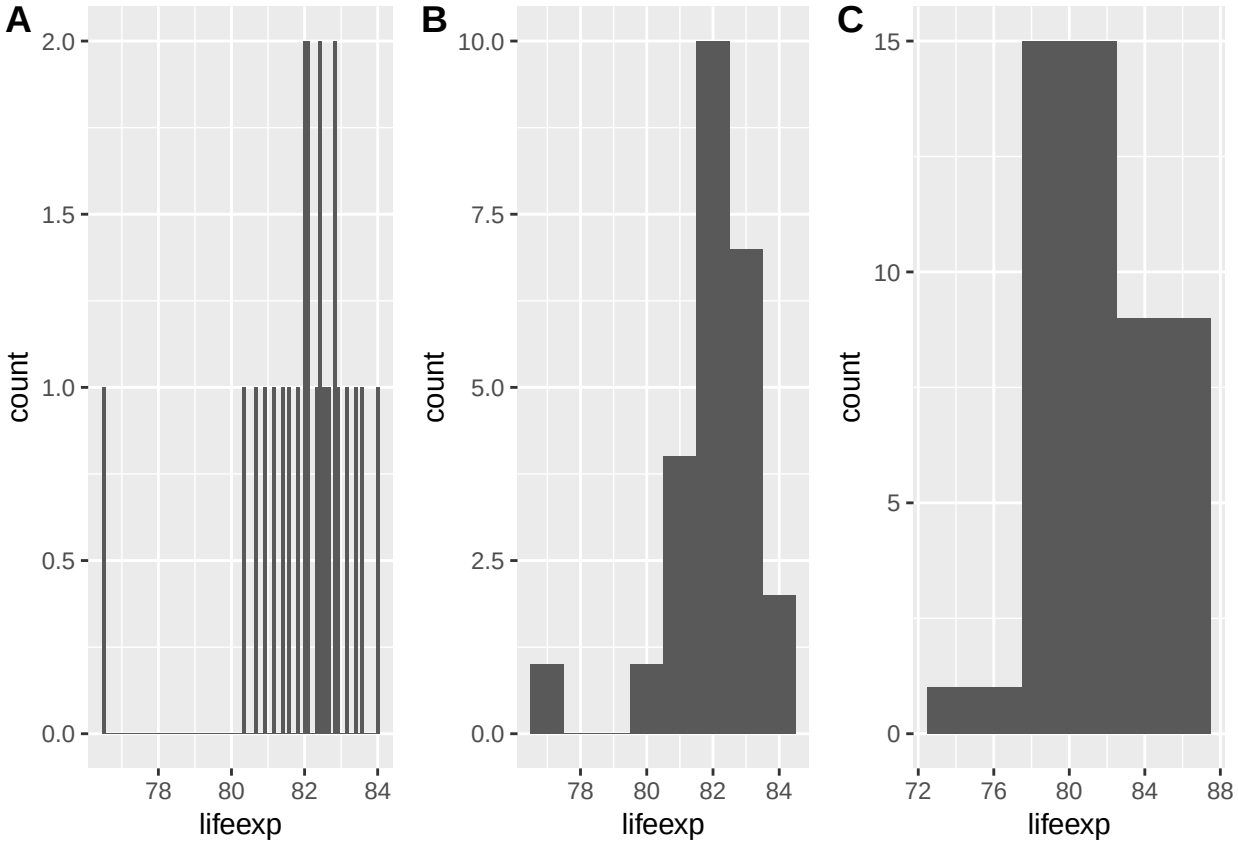
```
## Warning: Removed 18 rows containing non-finite values (stat_bin).
```




By default ggplot picks a binwidth of 1/30th the range of the data, yet **the optimal binwidth for a histogram depends on the range of your data, the number of observations, and the desired trade-off between detail and smoothing**. The fewer the observations, the bigger the bins need to be in order to create a usefull histogram. Say we want to show the distribution of life-expectancy among the 29 French NUTS-2 regions in 2015. Below you see 3 variations of doing that, using a binwidth of A) 1 month, B) half a year, and C) a year.

```
cowplot::plot_grid(

  euro_regio_eu %>%
  filter(year == 2015, country_code == 'FR') %>%
  ggplot() +
  geom_histogram(aes(x = lifeexp), binwidth = 1/12),

  euro_regio_eu %>%
  filter(year == 2015, country_code == 'FR') %>%
  ggplot() +
  geom_histogram(aes(x = lifeexp), binwidth = 1),

  euro_regio_eu %>%
  filter(year == 2015, country_code == 'FR') %>%
  ggplot() +
  geom_histogram(aes(x = lifeexp), binwidth = 5),

  labels = 'AUTO', ncol = 3

)
```

```
## Warning: Removed 4 rows containing non-finite values (stat_bin).

## Warning: Removed 4 rows containing non-finite values (stat_bin).

## Warning: Removed 4 rows containing non-finite values (stat_bin).
```



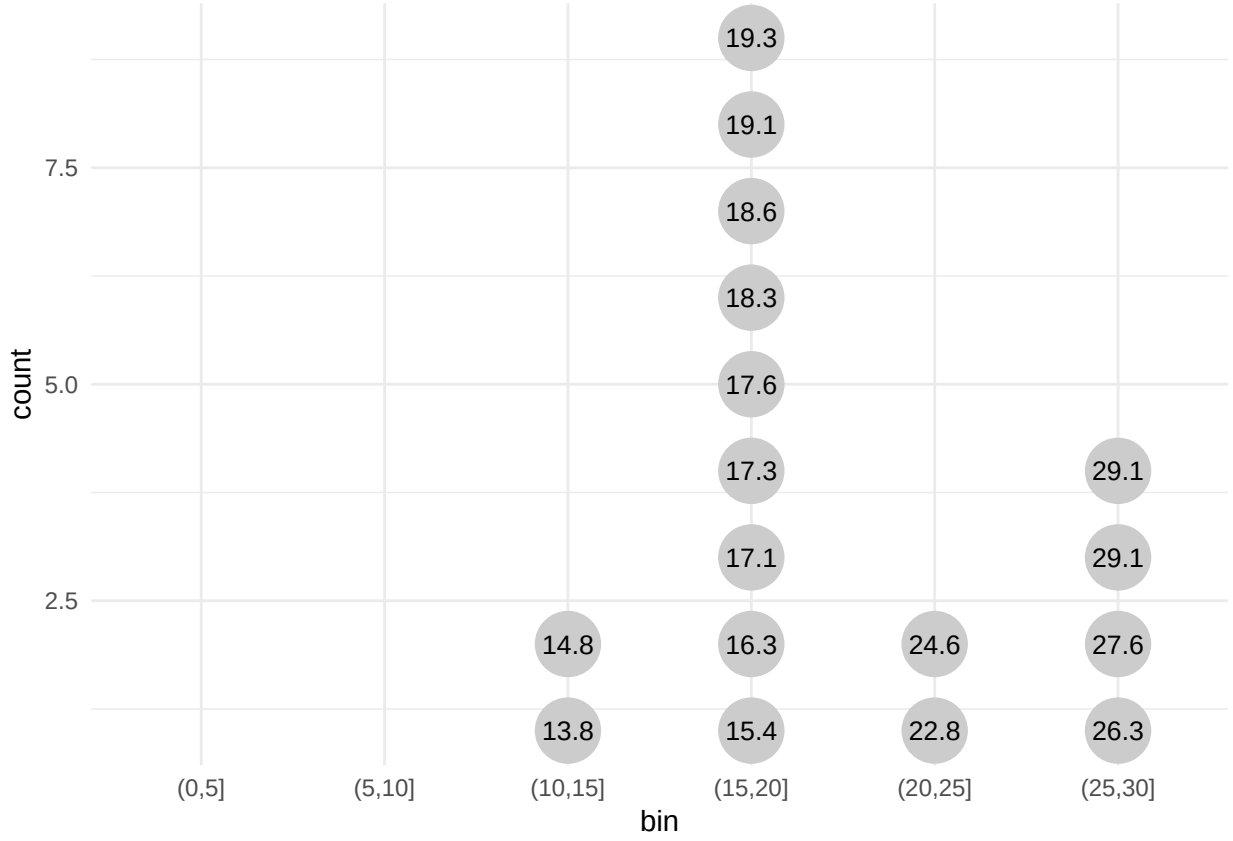

**Consider using a dot-histogram when ovservations are few (<100).** A dot-histogram[1] draws a single point for each observation, located at the center of the respective bin. This makes for a very intuitive visualization as the individual observations are still visible despite being aggregated into bins. Sadly the implementation of dot-histograms in ggplot is a bit lacking at the time of writing which necessitates to use our data wrangling skills to get a nice result.

```
# A trick I've learned from John Tukey's EDA: Label individual cases in
# dot-histograms. Aggregate and individual view at once!
# Here's how you do it in R: (link)

the_data <-
  euro_regio_eu %>%
  filter(year == 2015, country_code == 'ES') %>%
  select(id = nuts2_code, value = unemp)

# Label by id ------------------------------------------------------------

the_data %>%
  mutate(bin = cut(value, seq(0, 30, 5))) %>%
  na.omit() %>%
```

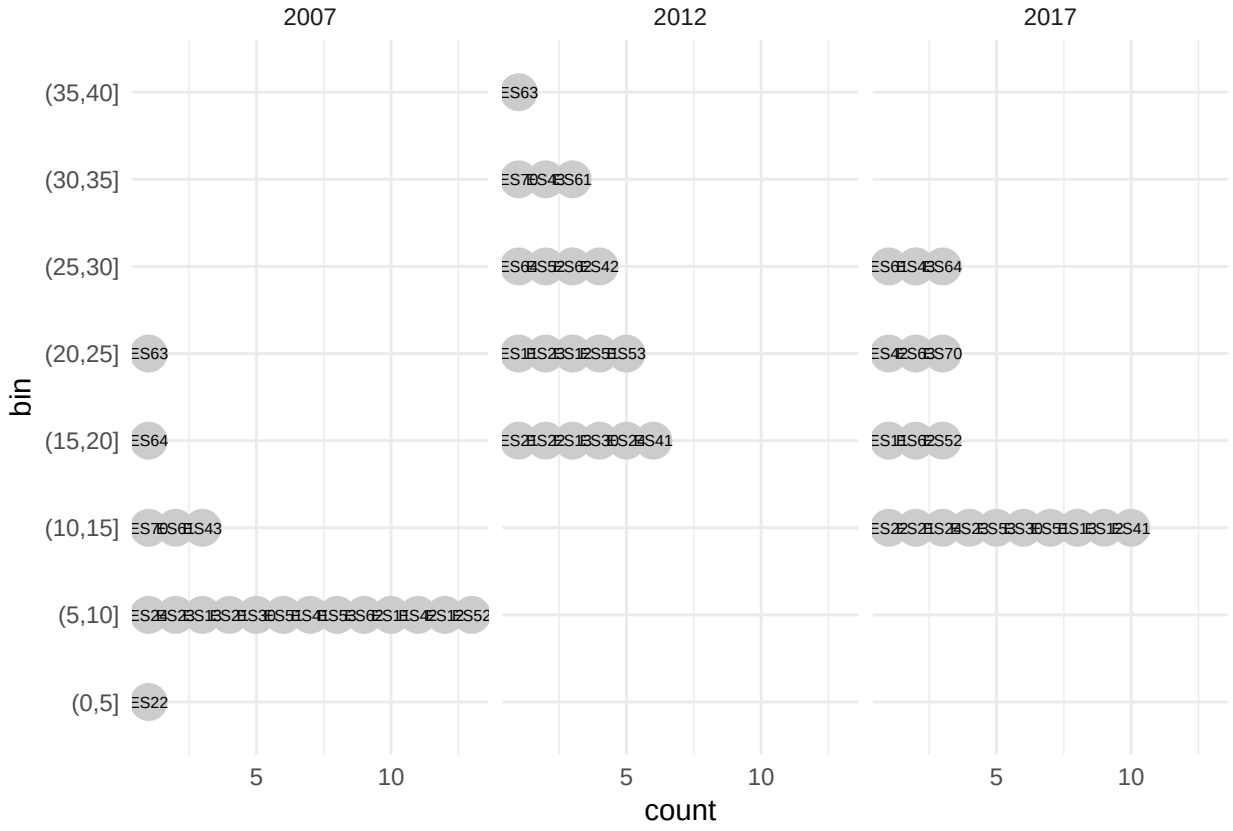
[1] Wilkinson, L. (1999). Dot Plots. The American Statistician, 53(3), 276. https://doi.org/10.2307/2686111

```
  group_by(bin) %>%
  arrange(value) %>%
  mutate(count = 1:n()) %>%
  ggplot(aes(x = bin, y = count)) +
  geom_point(aes(color = id), size = 11, color = 'grey80') +
  geom_text(aes(label = id), size = 3.4) +
  scale_x_discrete(drop = FALSE) +
  theme_minimal()
```


```
# Label by value ------------------------------------------------------------

 the_data %>%
  mutate(bin = cut(value, seq(0, 30, 5))) %>%
  na.omit() %>%
  group_by(bin) %>%
  arrange(value) %>%
  mutate(count = 1:n()) %>%
  ggplot(aes(x = bin, y = count)) +
  geom_point(aes(color = id), size = 11, color = 'grey80') +
  geom_text(aes(label = value), size = 3.4) +
  scale_x_discrete(drop = FALSE) +
  theme_minimal()
```




```
# Works with grouping as well ---------------------------------------------

the_data2 <-
  euro_regio_eu %>%
  filter(country_code == 'ES', year %in% c(2007, 2012, 2017)) %>%
  select(group = year, id = nuts2_code, value = unemp)

the_data2 %>%
  mutate(bin = cut(value, seq(0, 40, 5))) %>%
  na.omit() %>%
  group_by(group, bin) %>%
  arrange(value) %>%
  mutate(count = 1:n()) %>%
  ggplot(aes(x = bin, y = count)) +
  geom_point(aes(color = id), size = 6, color = 'grey80') +
  geom_text(aes(label = id), size = 2) +
  scale_x_discrete(drop = FALSE) +
  theme_minimal() +
  coord_flip() +
  facet_wrap(~group)
```




Whenever a `geom` uses a statistical transformation, such as the binning in `geom_histogram()`, a number of new variables are internally computed by `ggplot` prior to plotting. In case of `geom_histogram()` these new variables are

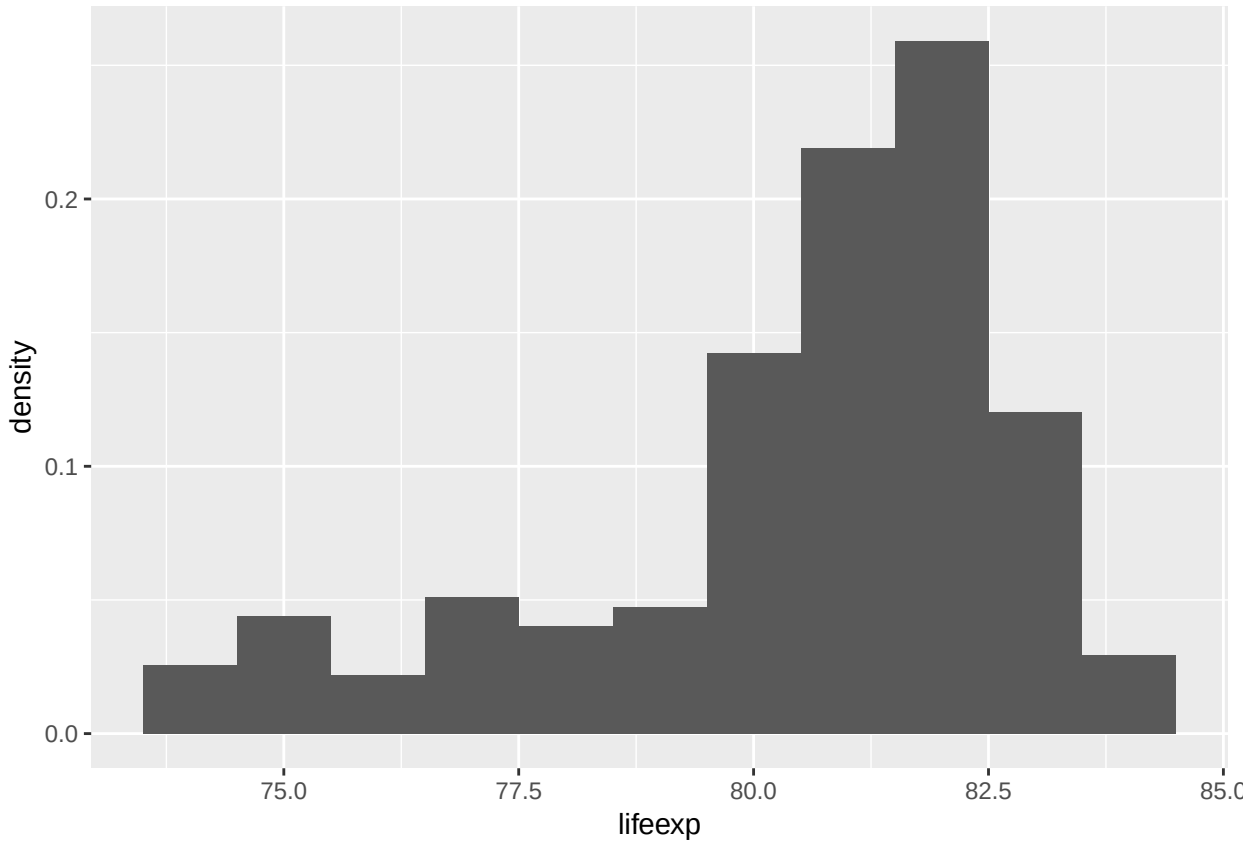
- `count`: number of points in bin
- `density`: density of points in bin
- `ncount`: count, scaled to a maximum of 1
- `ndensity`: density, scaled to a maximum of 1[2]

Via the `stat()` function we can use any of the computed variables in our plot. This makes it possible to draw histograms with `density` on the y axis instead of the default `count`.

```
euro_regio_eu %>%
  filter(year == 2015) %>%
  ggplot() +
  geom_histogram(aes(x = lifeexp, y = stat(density)), binwidth = 1)
```

```
## Warning: Removed 18 rows containing non-finite values (stat_bin).
```

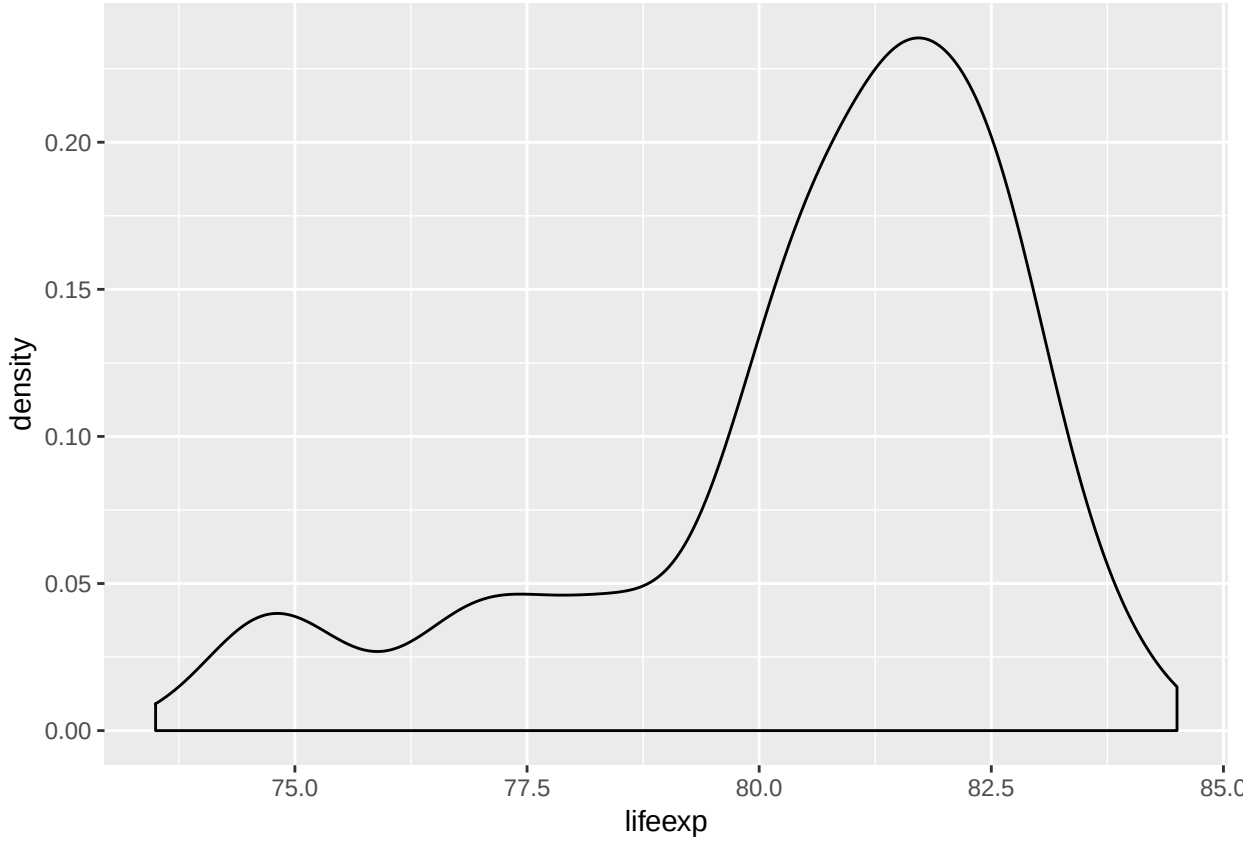
[2]you can find out about the computed variables by looking into the help page of a `geom`.




An alternative to the histogram is the **density plot**.

```
euro_regio_eu %>%
  filter(year == 2015) %>%
  ggplot() +
  geom_density(aes(x = lifeexp))
```

```
## Warning: Removed 18 rows containing non-finite values (stat_density).
```



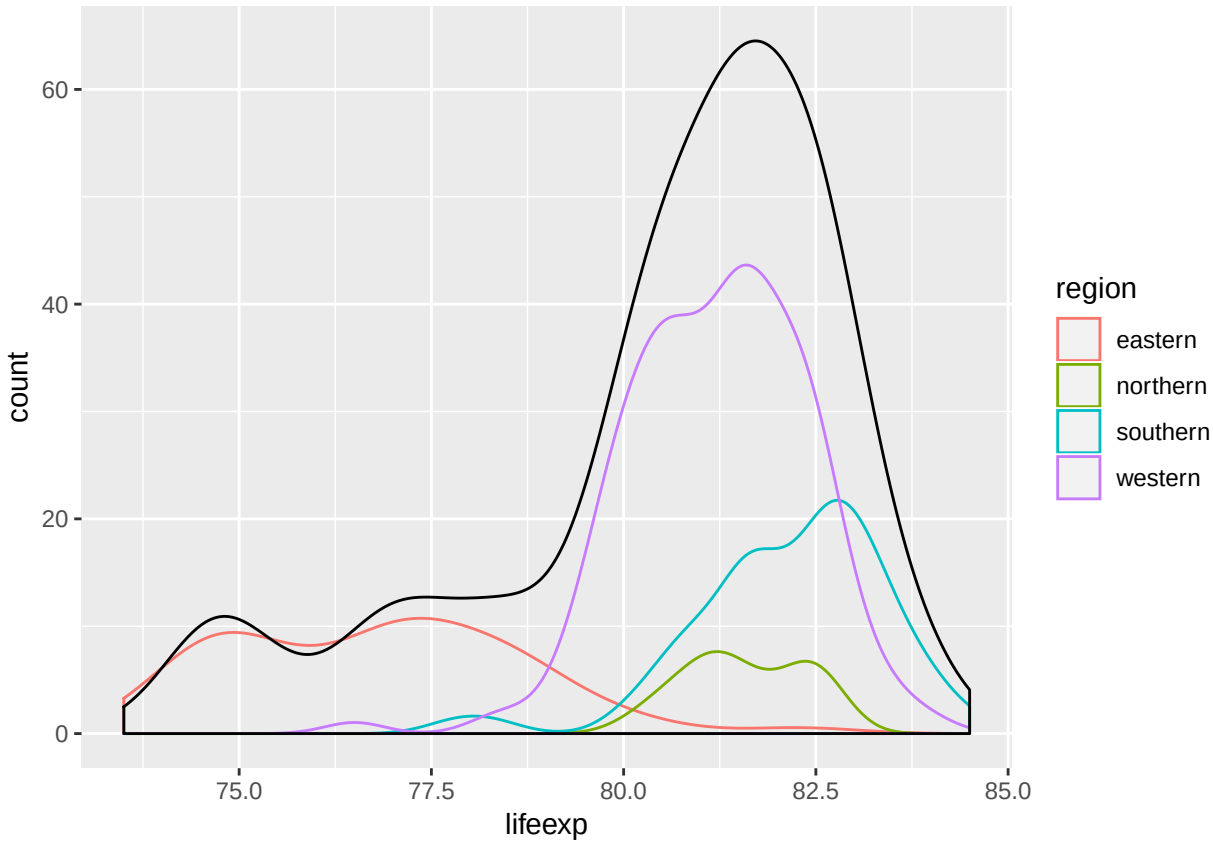

#### 4.5.1.2 Discrete

#### 4.5.1.3 Multivariate

```
euro_regio_eu %>%
  filter(year == 2015) %>%
  ggplot() +
  geom_density(aes(x = lifeexp, y = stat(count), color = region)) +
  geom_density(aes(x = lifeexp, y = stat(count)))
```

```
## Warning: Removed 18 rows containing non-finite values (stat_density).
```

```
## Warning: Removed 18 rows containing non-finite values (stat_density).
```




### 4.5.2 Visualizing association

#### 4.5.2.1 continuous vs. continuous

```
# scatterplot

# different symbols

# fill versus color

# white outline / overlap

# dot chart

# bubble chart

# dot density map
```

#### 4.5.2.2 continuous vs. discrete

#### 4.5.2.3 discrete vs. discrete

### 4.5.3 Visualizing change

#### 4.5.3.1 Lines

#### 4.5.3.2 Slope

#### 4.5.3.3 Connected dots

#### 4.5.3.4 Waterfall

#### 4.5.3.5 Bump

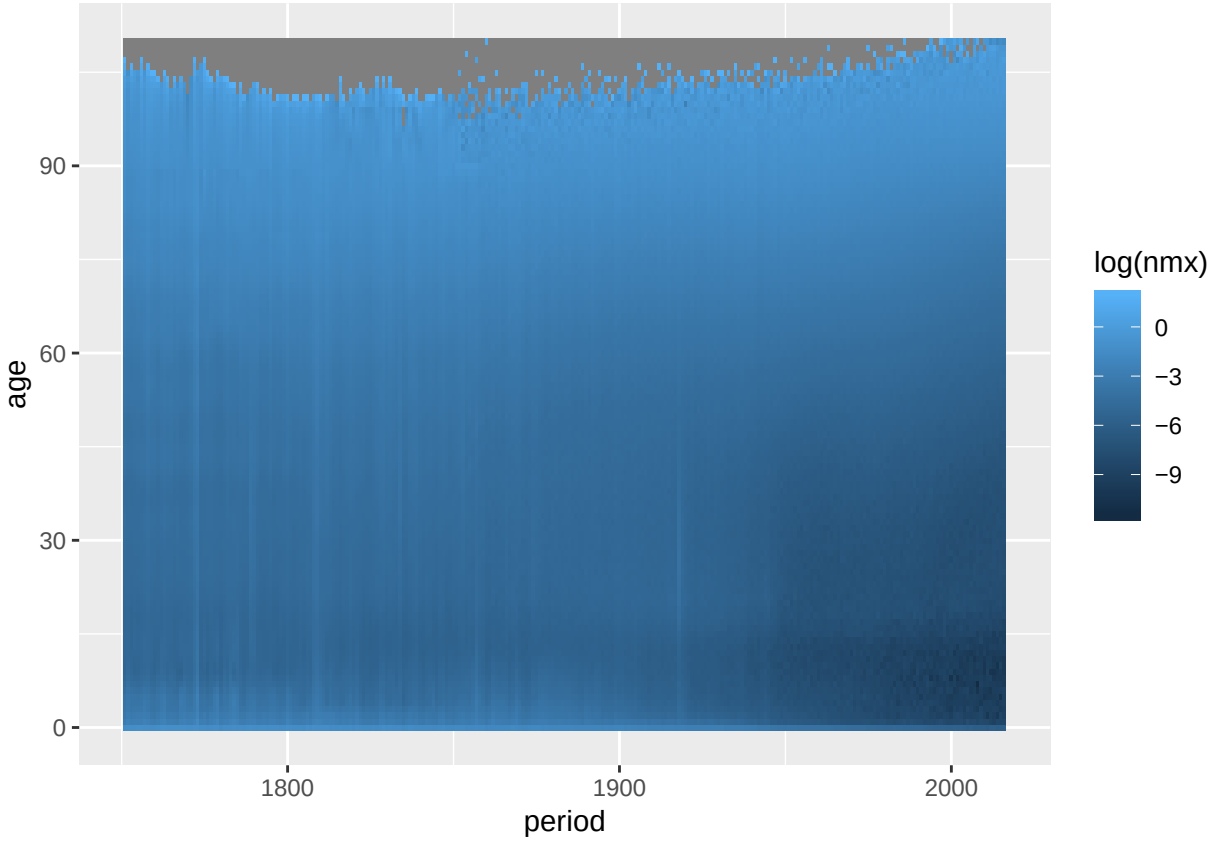
### 4.5.4 Visualizing surfaces

#### 4.5.4.1 Contours

#### 4.5.4.2 Heatmap

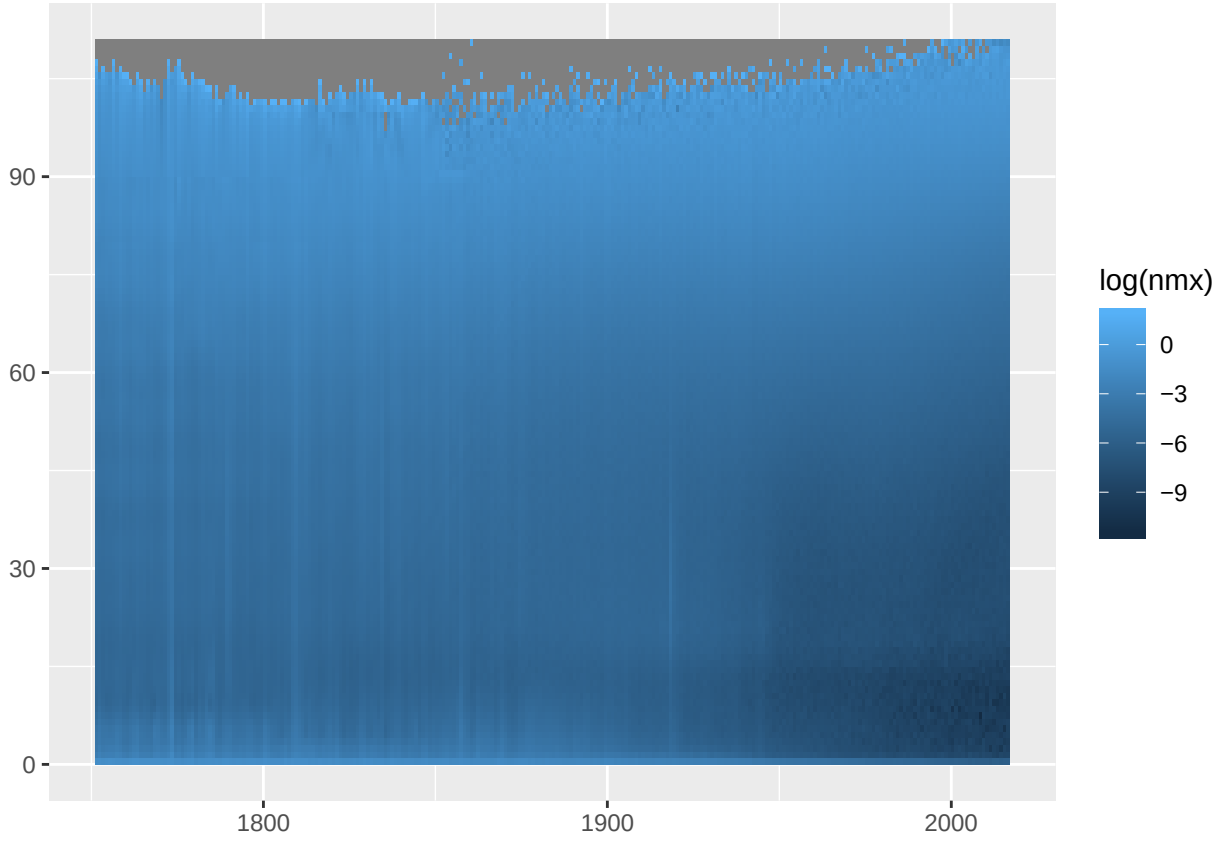
`geom_tile()` draws rectangles at specified xy-positions. A common use-case is to visualize the value of a variable across a surface by coloring the rectangles according to value. This plot type is also known as a *heatmap*. Demographers refer to heatmaps as *Lexis Surfaces* if the surface is defined by period/cohort and age. Here's an example showing Swedish age specific, logged death rates over time:

```
load('data/hmd/hmd_swe.RData')
swe <- mutate(hmd_swe, nmx = nDx/nEx)

ggplot(swe) +
  geom_tile(aes(x = period, y = age, fill = log(nmx)))
```




`geom_rect()` is an alternative to `geom_tile()` but we need to specify all four corners of each rectangle we wish to draw. As each age and period group is a single year wide, we specify rectangles which have a width and height of one unit.

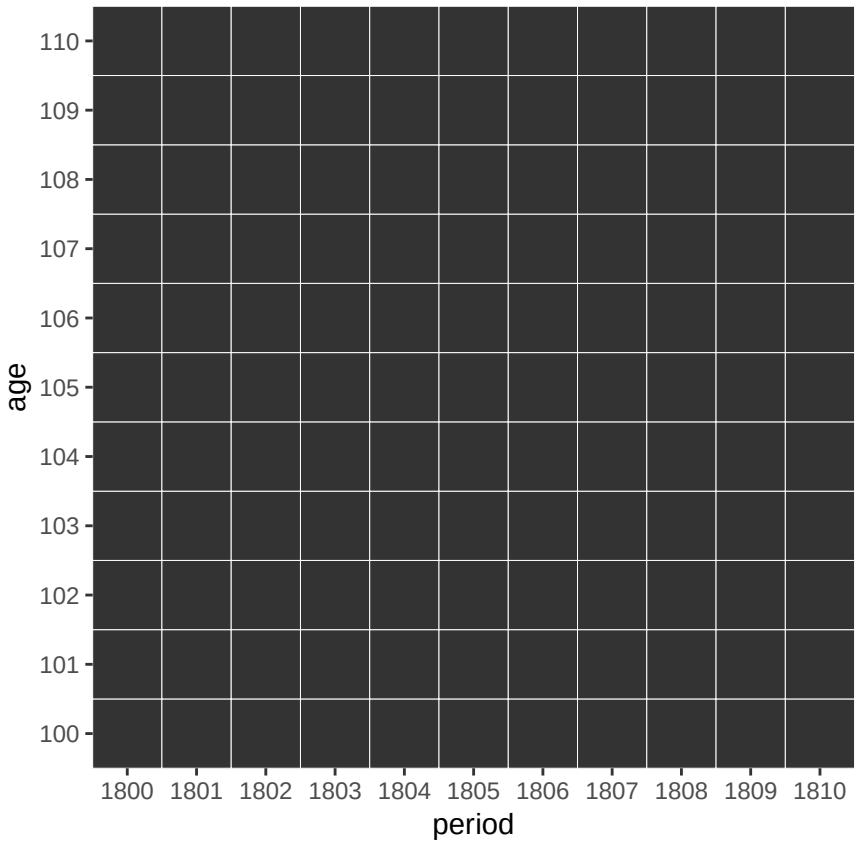
```
ggplot(swe) +
  geom_rect(aes(xmin = period, xmax = period+1, ymin = age, ymax = age+1,
                fill = log(nmx)))
```




Only specifying x and y position and omitting colour puts a grey rectangle at every xy position that appears in the data. The resulting plot gives us information about the period-ages where we have mortality data on Swedish females.

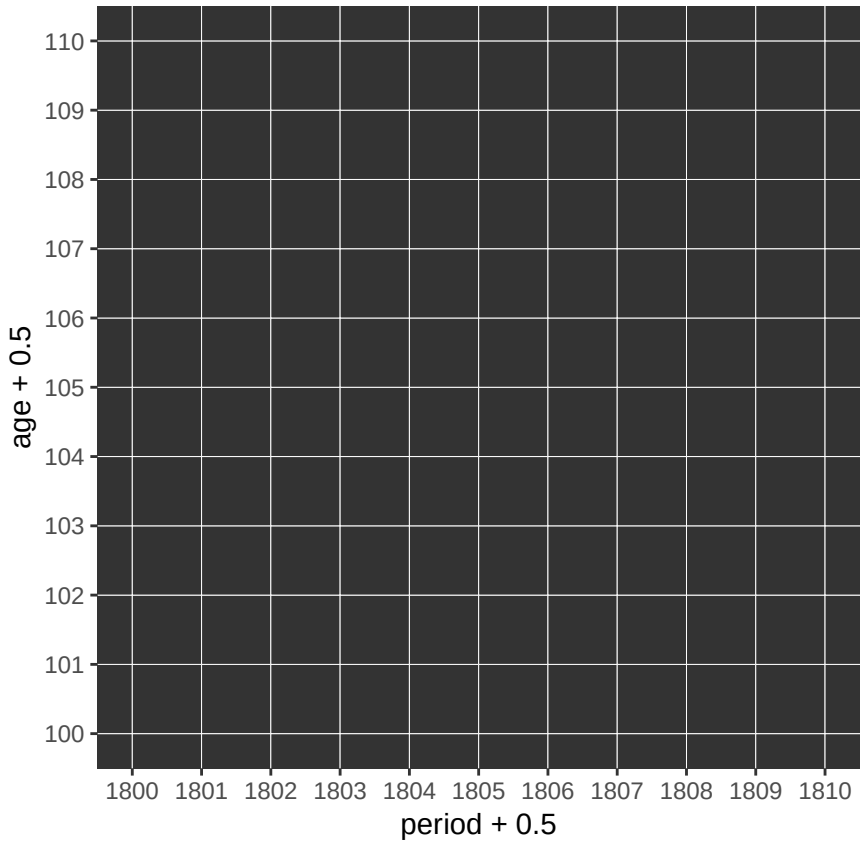
By default the small rectangles have a width and height of 1 scale unit and are drawn over the mid-points of the corresponding x and y values.

```
ggplot(swe) +
  geom_tile(aes(x = period, y = age), colour = "white") +
  scale_x_continuous(breaks = 1800:1810) +
  scale_y_continuous(breaks = 100:110) +
  coord_equal(xlim = c(1800, 1810), ylim = c(100, 110))
```




Shifting the data by 0.5 in x and y aligns things neatly.

```
ggplot(swe) +
  geom_tile(aes(x = period+0.5, y = age+0.5),
            colour = "white") +
  scale_x_continuous(breaks = 1800:1810) +
  scale_y_continuous(breaks = 100:110) +
  coord_equal(xlim = c(1800, 1810), ylim = c(100, 110))
```



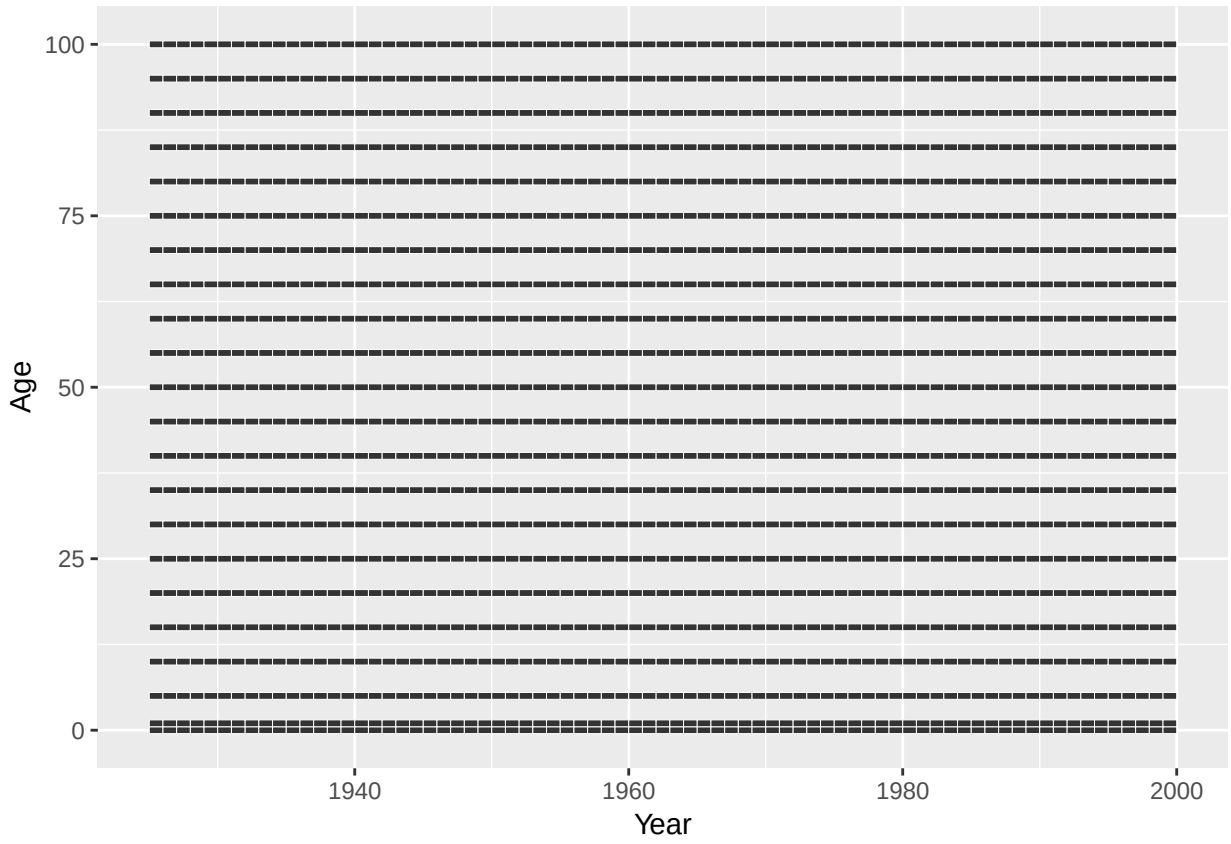

If our data does not come in 1 by 1 units we have to adjust the `width` and/or `height` of the rectangles. `width` and `height` are regular aesthetics and can be mapped to variables in the data.

```
cod <- read_csv('data/cod/cod.csv')
```

```
## Parsed with column specification:
## cols(
##   Year = col_integer(),
##   Age = col_integer(),
##   AgeGr = col_character(),
##   w = col_integer(),
##   Sex = col_character(),
##   COD = col_character()
## )
```

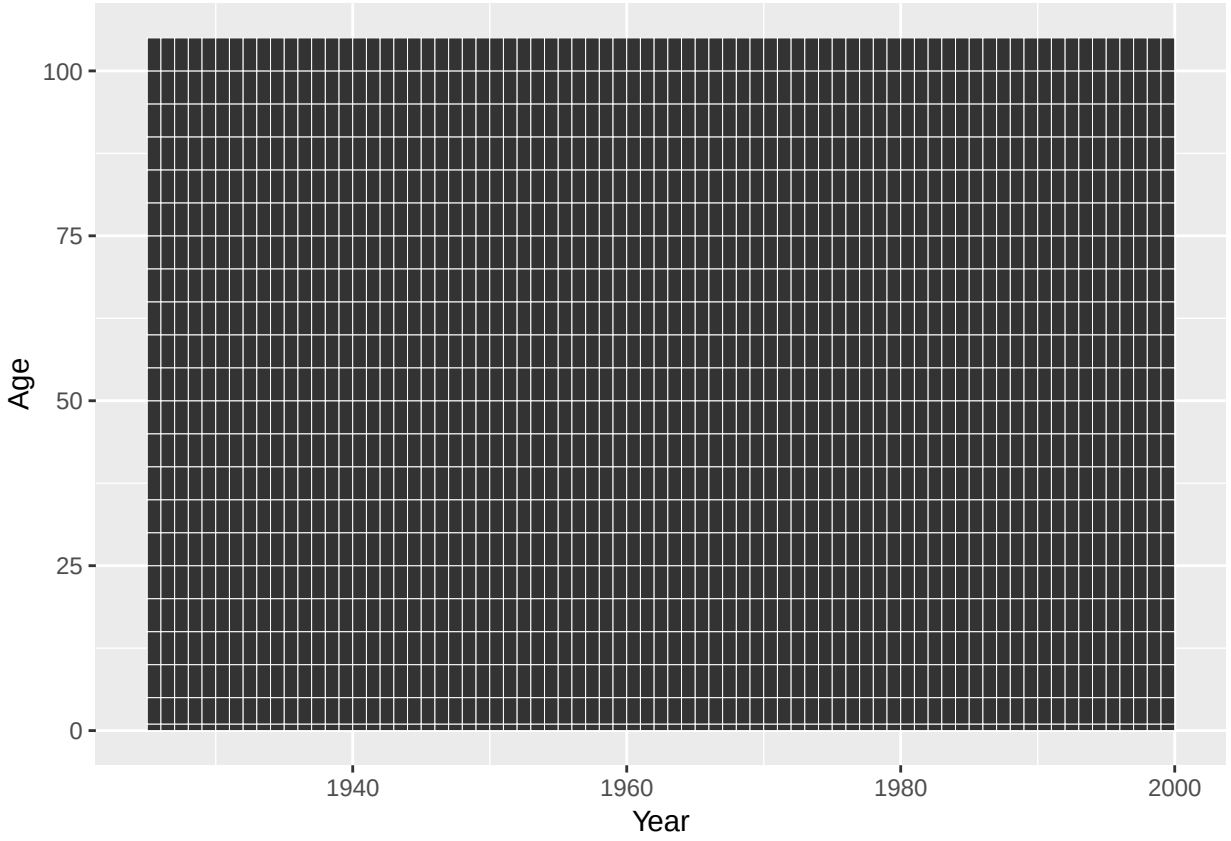
The Cause of Death data features age groups of different sizes (1, 4, or 5 years). This is how it looks like if we plot it without any regard to the size of the age groups.

```
cod %>% filter(Sex == "Female") %>%
  mutate(Year = Year + 0.5) %>%
  ggplot() +
  geom_tile(aes(x = Year, y = Age),
            colour = "white")
```



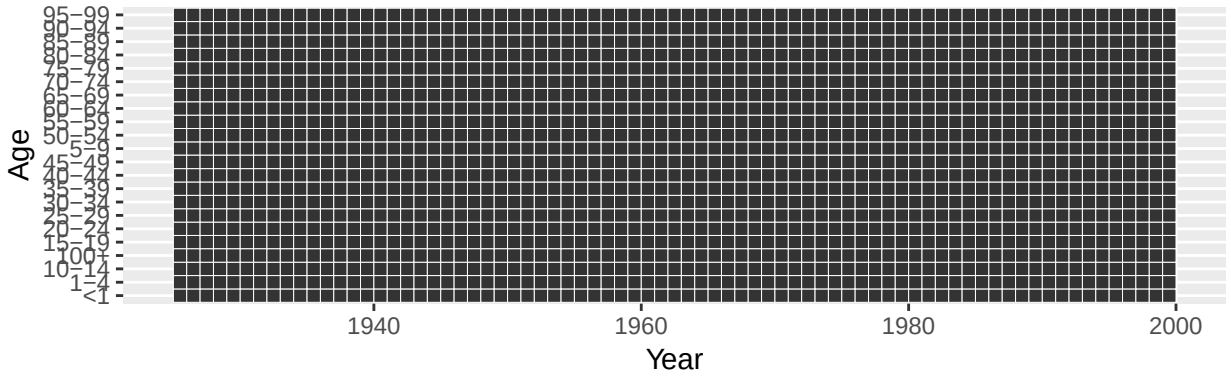

Now we shift the rectangles away from the age midpoint and scale them in height according to the width of the age group.

```
cod %>% filter(Sex == "Female") %>%
  mutate(Year = Year + 0.5, Age = Age + w/2) %>%
  ggplot() +
  geom_tile(aes(x = Year, y = Age, height = w),
            colour = "white")
```




If we use discrete axis (happens automatically if we supply a non-numeric variable to the x or y aesthetic) we loose any control over the placement of the age or period groups. They will be equally spaced along the axis.

```
cod %>% filter(Sex == "Female") %>%
  mutate(Year = Year + 0.5, Age = AgeGr) %>%
  ggplot() +
  geom_tile(aes(x = Year, y = Age), colour = "white") +
  coord_equal()
```



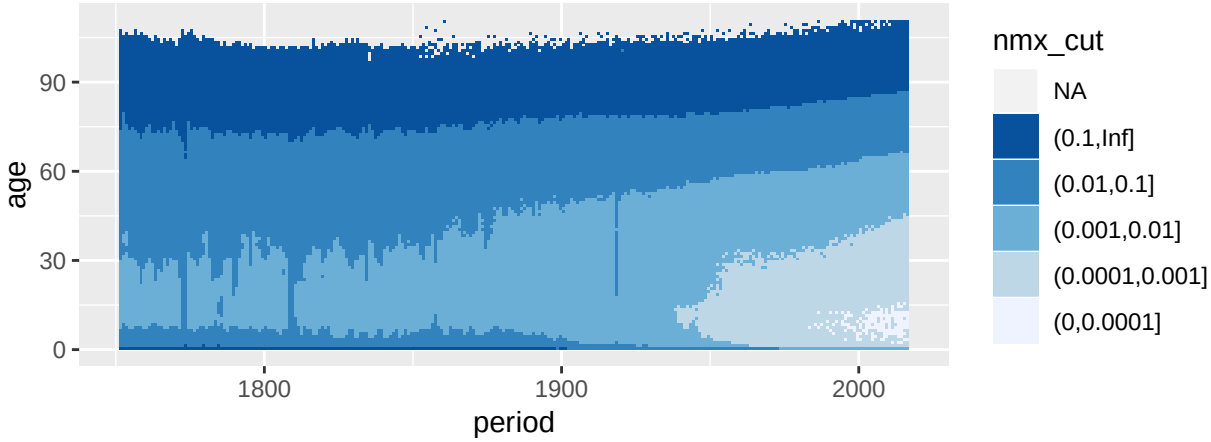

**Avoid character or factor variables as your period or age groups.** Whenever possible go with numeric "Start of Interval" and "Interval Width" variables.

#### 4.5.4.2.1 Sequential Colour Scales: Plotting Magnitudes

If we plot magnitudes we would like to use a colour scale which has an intrinsic ordering to it. Scales that

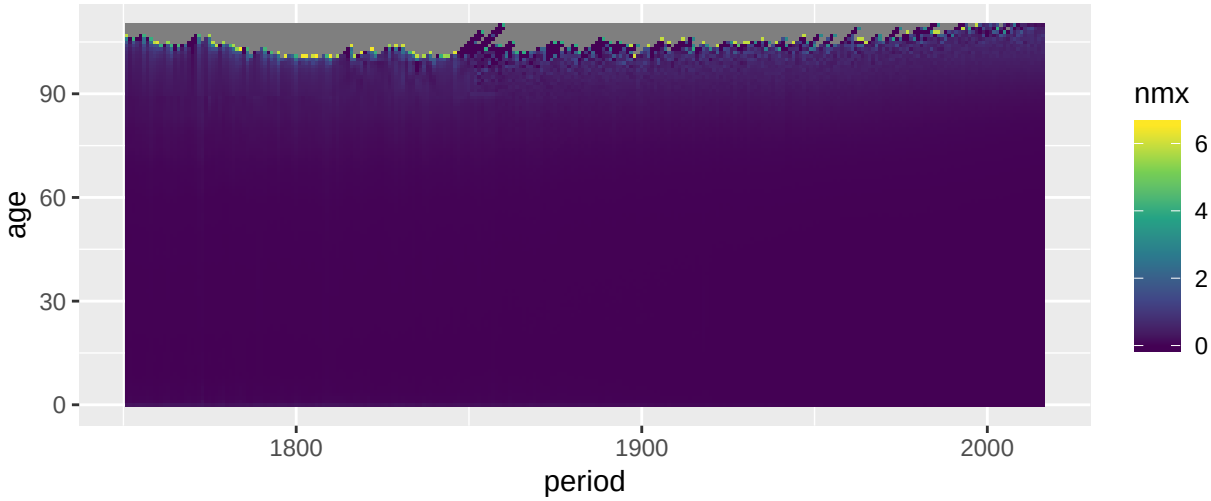
vary from dark to light are suitable and we call them "sequential". `scale_fill_brewer(type = "seq")` provides you with such a scale.

```
breaks_nmx <- c(0, 0.0001, 0.001, 0.01, 0.1, Inf)
swe %>%
  mutate(nmx_cut = cut(nmx, breaks = breaks_nmx)) %>%
  ggplot() +
  geom_tile(aes(x = period, y = age, fill = nmx_cut),
            position = position_nudge(0.5, 0.5)) +
  scale_fill_brewer(type = "seq") +
  guides(fill = guide_legend(reverse = TRUE)) +
  coord_equal()
```




Continuous colour scales take the form of a smooth colour gradient. Getting the gradient to look like you want can be tricky.

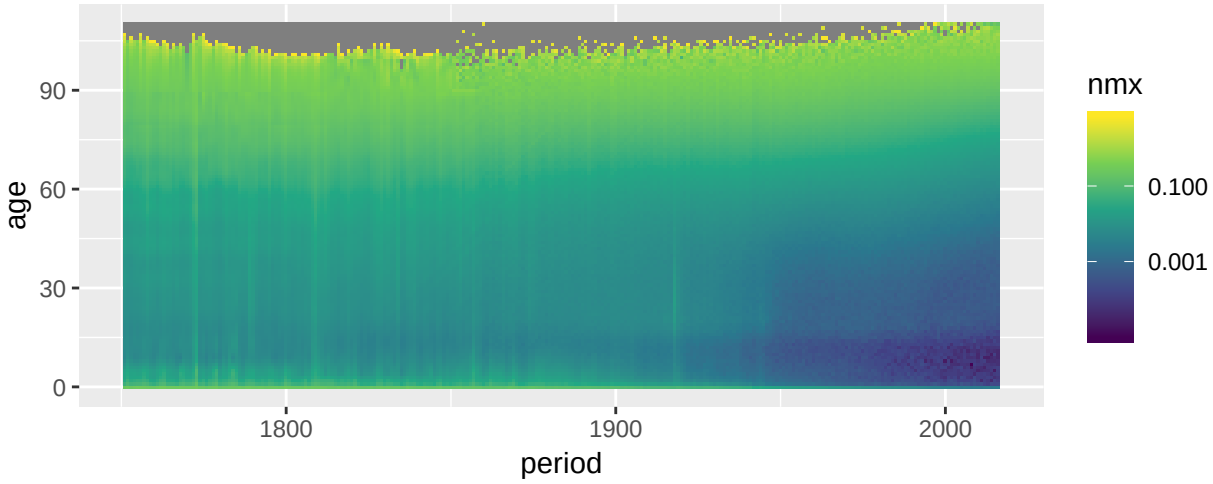
```
swe %>%
  ggplot() +
  geom_tile(aes(x = period, y = age, fill = nmx)) +
  scale_fill_viridis_c() +
  coord_equal()
```




Log transform the colour scale.

```
swe %>%
  ggplot() +
  geom_tile(aes(x = period, y = age, fill = nmx)) +
  scale_fill_viridis_c(trans = 'log10') +
  coord_equal()
```

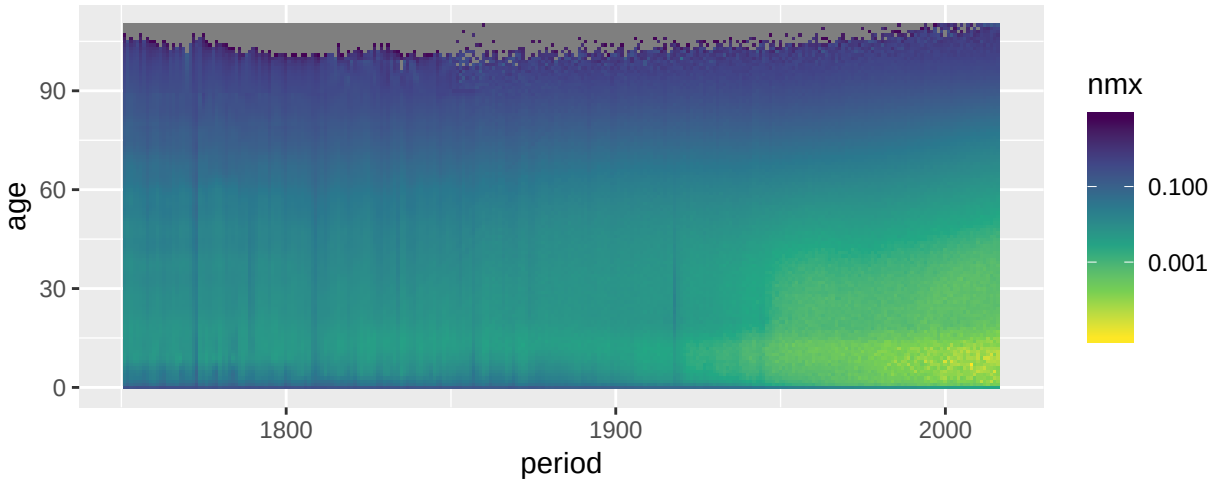
```
## Warning: Transformation introduced infinite values in discrete y-axis
```




Make high values dark and low values light.

```
swe %>%
  ggplot() +
  geom_tile(aes(x = period, y = age, fill = nmx)) +
  scale_fill_viridis_c(trans = 'log10',
                       direction = -1) +
  coord_equal()
```

```
## Warning: Transformation introduced infinite values in discrete y-axis
```

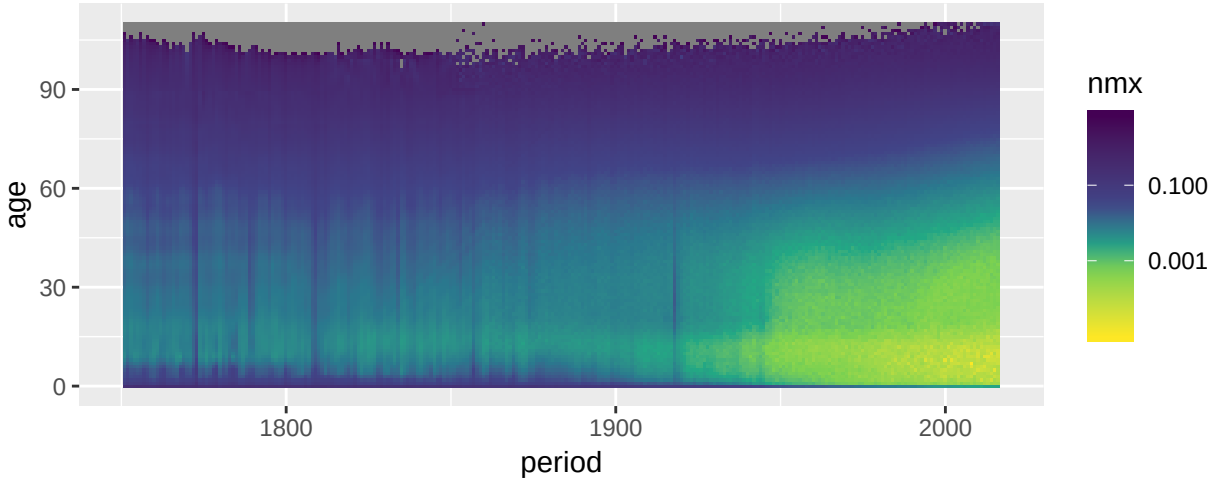



Rescale the colour gradient to increase contrast.

```
swe %>%
  ggplot() +
  geom_tile(aes(x = period, y = age, fill = nmx)) +
```

```
  scale_fill_viridis_c(trans = 'log10',
                       direction = -1,
                       values = c(0, 0.3, 0.4, 0.5, 0.6, 1)) +
  coord_equal()
```

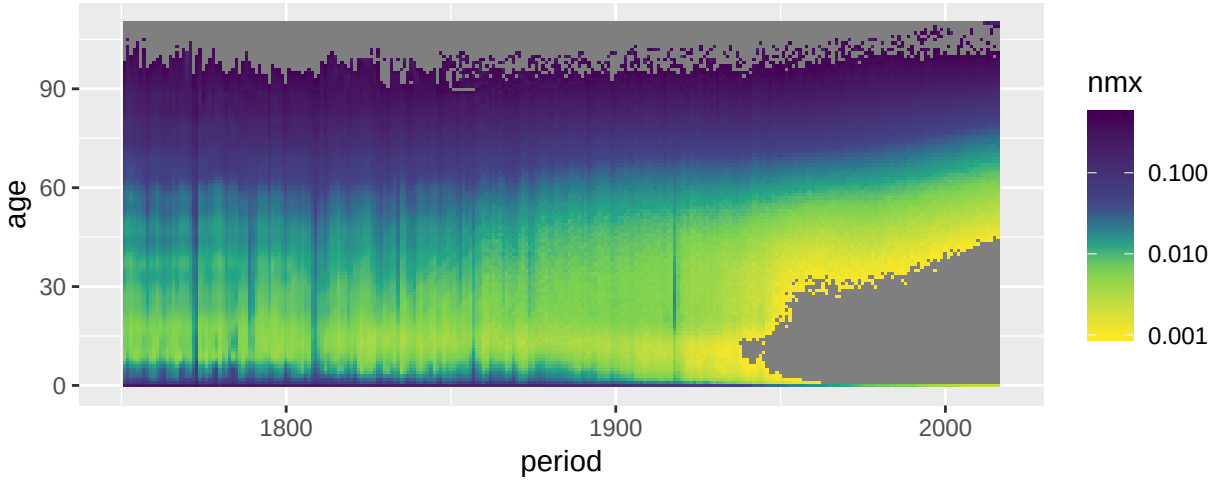
```
## Warning: Transformation introduced infinite values in discrete y-axis
```




Throw away data outside of the limits.

```
swe %>%
  ggplot() +
  geom_tile(aes(x = period, y = age, fill = nmx)) +
  scale_fill_viridis_c(trans = 'log10',
                       direction = -1,
                       values = c(0, 0.3, 0.4, 0.5, 0.6, 1),
                       limits = c(0.001, 0.5)) +
  coord_equal()
```

```
## Warning: Transformation introduced infinite values in discrete y-axis
```

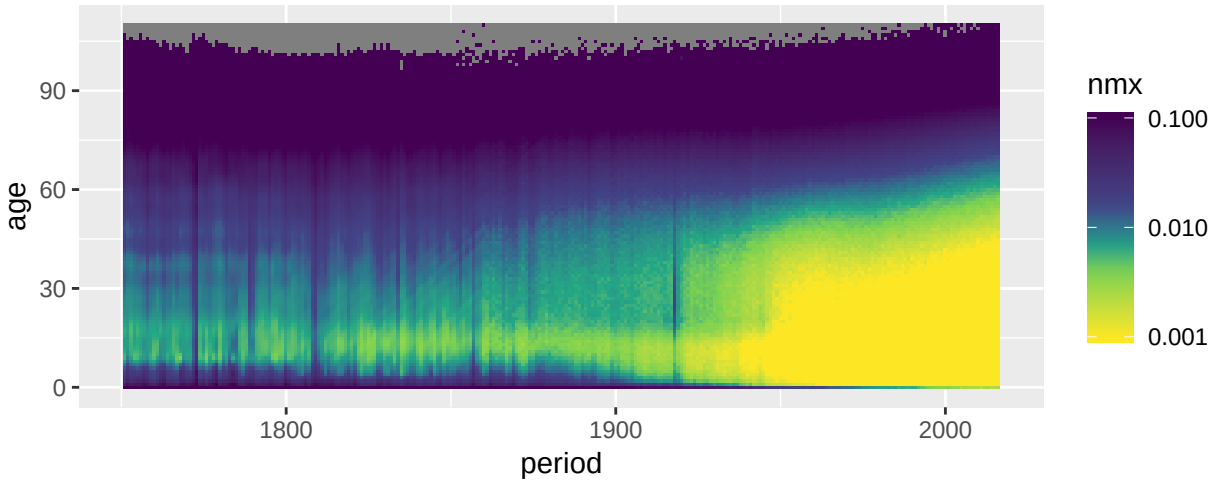



Or instead *squish* the out-of-bounds data into the limits.

```
swe %>%
  ggplot() +
```

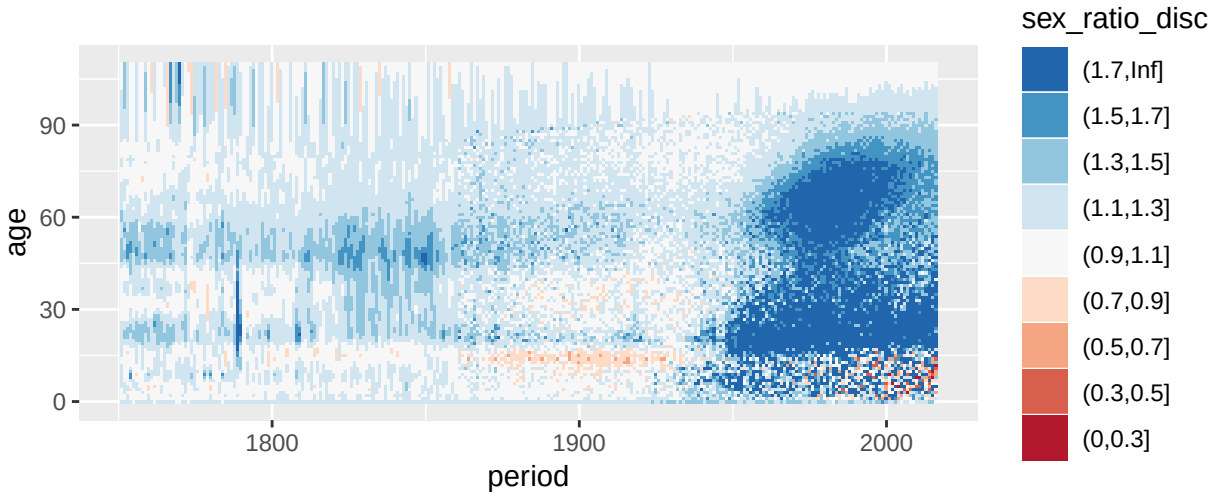
```
  geom_tile(aes(x = period, y = age, fill = nmx)) +
  scale_fill_viridis_c(trans = 'log10',
                       direction = -1,
                       values = c(0, 0.3, 0.4, 0.5, 0.6, 1),
                       limits = c(0.001, 0.1),
                       oob = scales::squish) +
  coord_equal()
```

```
## Warning: Transformation introduced infinite values in discrete y-axis
```



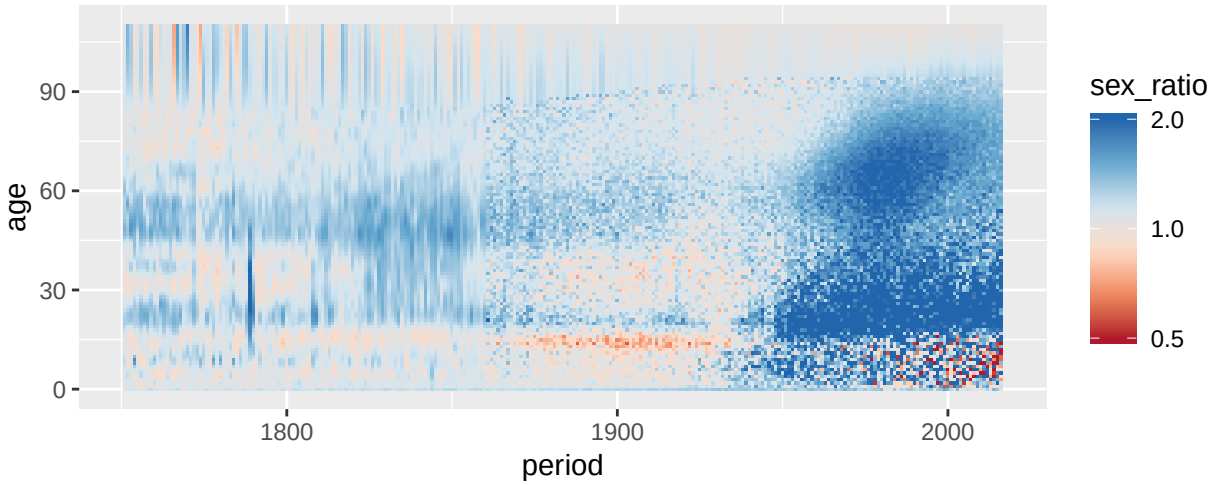

#### 4.5.4.2.2 Divergent colour scales: Plotting Differences & Proportions

```
load('data/hmd/hmd.RData')
breaks_prop_nmx <- c(0, 0.3, 0.5, 0.7, 0.9,
                     1.1, 1.3, 1.5, 1.7, Inf)

hmd %>%
  filter(country == 'SWE') %>%
  select(sex, period, age, nmx) %>%
  spread(sex, nmx) %>%
  mutate(sex_ratio = Male/Female,
         sex_ratio_disc = cut(sex_ratio, breaks_prop_nmx)) %>%
  ggplot() +
  geom_tile(aes(x = period, y = age, fill = sex_ratio_disc)) +
  scale_fill_brewer(type = "div",
                    palette = 5) +
  guides(fill = guide_legend(reverse = TRUE)) +
  coord_equal()
```




Continuous variant.

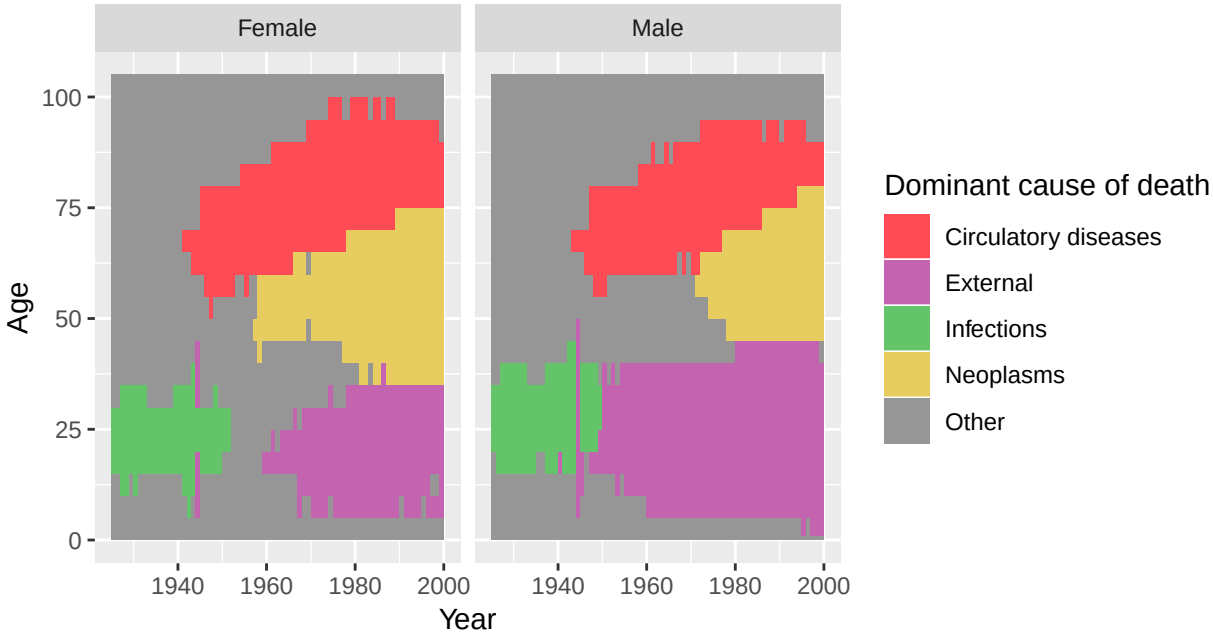
```
hmd %>%
  filter(country == 'SWE') %>%
  select(sex, period, age, nmx) %>%
  spread(sex, nmx) %>%
  mutate(sex_ratio = Male/Female) %>%
  ggplot() +
  geom_tile(aes(x = period, y = age, fill = sex_ratio)) +
  # takes 6 colours from a brewer palette and interpolates
  scale_fill_distiller(type = "div",
                       palette = "RdBu",
                       trans = "log2",
                       limits = c(0.5, 2),
                       direction = 1,
                       oob = scales::squish) +
  coord_equal()
```




#### 4.5.4.2.3 Qualitative colour scales: Plotting Group Membership

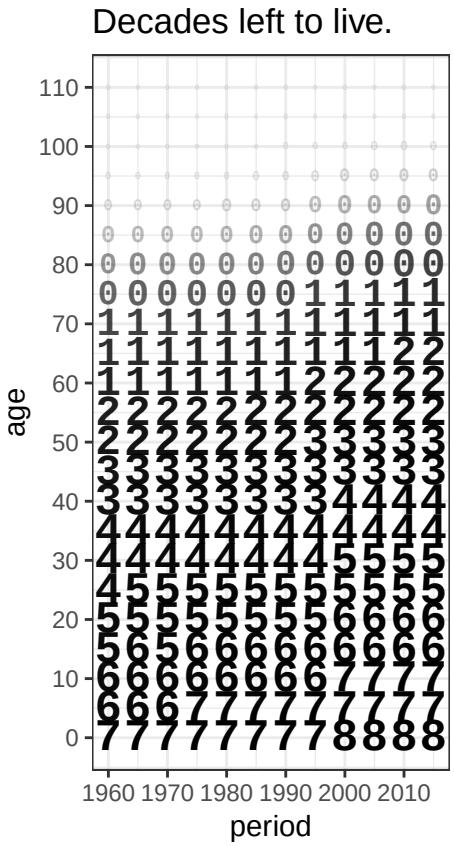
```
cod %>%
  mutate(Year = Year + 0.5, Age = Age + w/2) %>%
```

```
  ggplot() +
  geom_tile(aes(x = Year, y = Age, height = w, fill = COD)) +
  coord_equal() +
  facet_wrap(~Sex, ncol = 2) +
  scale_fill_manual("Dominant cause of death",
                    values = c("Infections" = "#63C46A",
                               "Neoplasms" = "#E7CC60",
                               "Circulatory diseases" = "#FF4B56",
                               "External" = "#C463AF",
                               "Other" = "#969696"))
```

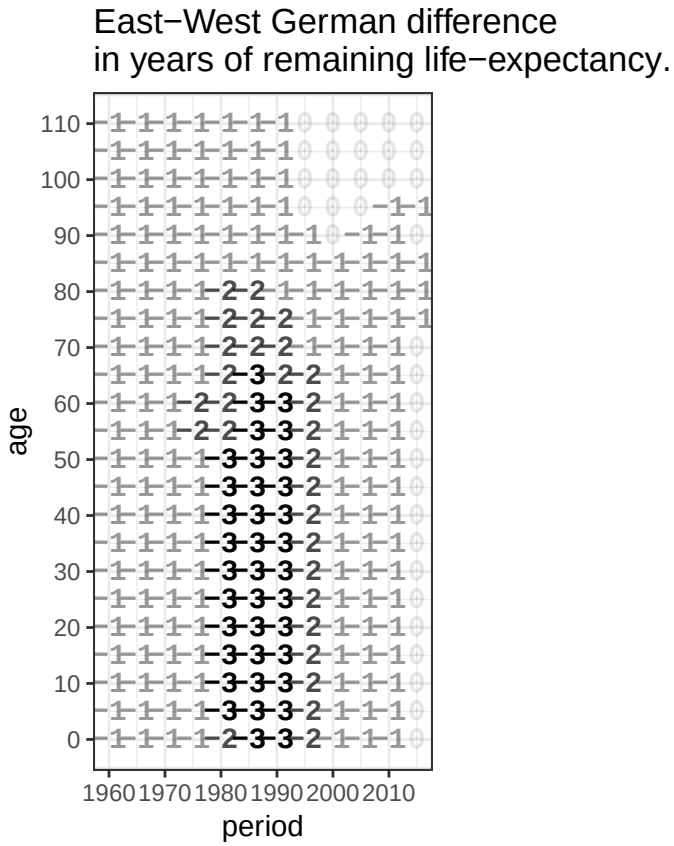



#### 4.5.4.3 Tables

```
hmd %>%
  filter(country == 'DEUTE', sex == 'Female',
         (period %% 5) == 0, (age %% 5) == 0) %>%
  ggplot(aes(x = period, y = age, alpha = lx, size = lx)) +
  geom_text(aes(label = ex%/%10),
            family = 'mono', fontface = 'bold',
            show.legend = FALSE) +
  scale_x_continuous(breaks = seq(1900, 2010, 10)) +
  scale_y_continuous(breaks = seq(0, 110, 10)) +
  theme_bw() +
  coord_equal() +
  labs(title = 'Decades left to live.')
```




```
hmd %>%
  filter(country %in% c('DEUTE', 'DEUTW'), sex == 'Female',
         (period %% 5) == 0, (age %% 5) == 0) %>%
  select(country, period, age, ex) %>%
  spread(country, ex) %>%
  mutate(d_ex = DEUTE - DEUTW) %>%
  ggplot(aes(x = period, y = age)) +
  geom_text(aes(label = d_ex%/%1, alpha = abs(d_ex%/%1)),
            family = 'mono', fontface = 'bold',
            show.legend = FALSE) +
  scale_x_continuous(breaks = seq(1900, 2010, 10)) +
  scale_y_continuous(breaks = seq(0, 110, 10)) +
  theme_bw() +
  coord_equal() +
  labs(title = 'East-West German difference\nin years of remaining life-expectancy.')
```




### 4.5.5 Excercise: Chart type vocabulary

1) Using `ggplot2`, plot a population pyramid using data of your choice.
2) Calculate the differences of Danish and Swedish remaining male life-expectancies by period and age. Visualize the result as a Lexis surface with a discrete divergent color scale. The choice of breaks is up to you.

## 4.6 Polish your plot for publication

### 4.6.1 Highlight

### 4.6.2 Annotate

## 4.7 Maps

## 4.8 Networks