KOSTAT-UNFPA Summer Seminar on Population

Workshop 1. Demography in R

Day 2: The tidy data approach

Instructor: Tim Riffe
`tim.riffe@ehu.eus`

Assistants:
Jinyeon Jo: `jyjo43043@gmail.com`
Rustam Tursun-Zade: `rustam.tursunzade@gmail.com`

28 July 2022

Contents

1 Tidy data

1.1 Definition

Tidy data follows a standard structure where each column is a variable, each row is an observation, and each cell is a value. Anything else is messy. It's literally that straightforward. A more complete definition can be found here: https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html Demographic data is often delivered in a tidy format. When it is not, then it can be reshaped into a tidy format.

Tidyverse packages work well together because they share a standard approach to formatting and working with datasets. Tidy datasets processed using tidyverse tools allow for fast and understandable analyses that in many cases require no *programming*, whereas it often takes a certain amount of head-scratching (programming) to analyze not-tidy datasets.

Tidy datasets can also be visualized without further ado using a systematic *grammar* (Wilkinson 2012) implemented in the `ggplot2` package (Wickham (2016), this loads automatically with `tidyverse`). Today we will do just basic examples, but this will be made more explicit as the workshop progresses.

1.2 Example (gapminder)

The so-called `gapminder` dataset is an example of *tidy* data that allows to demonstrate some of the basic `tidyverse` concepts. Let's install this package and have a look. Remember to comment out the installation line of code using `#` after you install it once!

```
install.packages("gapminder")
```

```
library(gapminder)
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.7     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
?gapminder
#View(gapminder)
# list the data structure:
str(gapminder)
```

```
## tibble [1,704 x 6] (S3: tbl_df/tbl/data.frame)
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
```

```
##  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
##  $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 12881816 1
##  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

In this data, unique combinations of country and year are what define an observation. From the above call to `str()` we see the structure of the data, which indicates the column types and the number of rows (1704). We therefore have 1704 observations. `continent` is a property of `country` here, and is not a structural variable.

We have three *variables* spread over the columns, life expectancy at birth `lifeExp`, population size `pop`, and GDP per capita `gdpPercap`.

1.3 Basic dataset descriptives

There is a function called `summary()` that guesses how we would like the data summarized:

```
summary(gapminder)
```

```
##         country        continent        year         lifeExp
##  Afghanistan:  12   Africa  :624   Min.   :1952   Min.   :23.60
##  Albania    :  12   Americas:300   1st Qu.:1966   1st Qu.:48.20
##  Algeria    :  12   Asia    :396   Median :1980   Median :60.71
##  Angola     :  12   Europe  :360   Mean   :1980   Mean   :59.47
##  Argentina  :  12   Oceania : 24   3rd Qu.:1993   3rd Qu.:70.85
##  Australia  :  12                  Max.   :2007   Max.   :82.60
##  (Other)    :1632
##       pop               gdpPercap
##  Min.   :6.001e+04   Min.   :   241.2
##  1st Qu.:2.794e+06   1st Qu.:  1202.1
##  Median :7.024e+06   Median :  3531.8
##  Mean   :2.960e+07   Mean   :  7215.3
##  3rd Qu.:1.959e+07   3rd Qu.:  9325.5
##  Max.   :1.319e+09   Max.   :113523.1
##
```

The result tells us that there are 12 observations for each country, that there are 624 observations in Africa, 300 in the Americas, etc, and it usefully gives the range and quartiles of each variable. For example life expectancy observations in the data range from 23.6 to 82.6. Wow!

One can also query specific columns like so: We can check the year range like so:

```
unique(gapminder$year)
```

```
##  [1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
```

```
# or
gapminder %>% pull(year) %>% unique()
```

```
##  [1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
```

This data is in 5-year intervals, and year values appear to be approximately centered within standard intervals. i.e. 1950-1954 gets the value 1952. We have about 50 years of history here. We give two equivalent ways of asking this question of the data. The second is easier to deparse visually, even if we've not yet introduced the operator `%>%` or the function `pull()`. It reads "take the gapminder data, then pull off the year column, then give the unique values". The symbol `%>%` reads as "then". The expression `unique(gapminder$year)` on the other hand is somehow inverted, meaning that it reads from the inside out. We start with the year column of gapminder, then look outward to see that we extract its unique values. Both are valid approaches. The one using pipes is the version we will more often attempt to use in this workshop.

1.3.1 Mini exercise

List which countries are in the data, and write how many there are:

```
#
```

1.4 pipes

The pipe operator, more explicitly works by evaluating an object on the left and sending the result to the function on the right.

For example, the below pipe separates step 1 (the drawing of 10 random deviates of the uniform distribution) from step 2 (calculation of their mean).

```
runif(10) %>% mean()
```

```
## [1] 0.5799435
```

One can chain together a sequence of operations like so:

```
runif(10) %>%
  sort() %>%
  cumsum()
```

```
##  [1] 0.07807535 0.29046070 0.80297450 1.32949541 2.09031763 2.88301427
##  [7] 3.73432112 4.65528704 5.59519759 6.55659670
```

This code reads in order "take ten random uniform draws, then sort them (in ascending order), then calculate their cumulative sum". Let's call this sort of code statement a *pipeline*, since it defines a multistep sequence of execution steps. We will be construction data analysis sequences using this trick for the entirety of the workshop. If it is not immediately clear what is happening here, do not worry, it will make sense as we progress through the material, and I will redundantly narrate each code chunk multiple times.

1.4.1 Mini Exercise for pipes

Take 100 random draws of the Poisson distribution, with lambda parameter equal to 100 (`rpois()`), and calculate the 95% prediction interval using `quantile(x, probs = c(.025,.975))`. Note that the argument `x` is simply going to be the incoming data from `rpois()`, and you don't need to specify the argument `x` at all.

```
#
```

I introduce this now, so that we may use it naturally in what comes.

1.5 filtering is for rows

Filtering in the `tidyverse` implies the potential deletion of rows based on some logical criteria. Observe:

```
A <- tibble(a = 0:10,
            b = letters[1:11])
A
```

```
## # A tibble: 11 x 2
##         a b
```

```
##    <int> <chr>
##   1      0 a
##   2      1 b
##   3      2 c
##   4      3 d
##   5      4 e
##   6      5 f
##   7      6 g
##   8      7 h
##   9      8 i
##  10      9 j
##  11     10 k
```

```r
A %>%
  filter(a > 5)
```

```
## # A tibble: 5 x 2
##       a b
##   <int> <chr>
## 1     6 g
## 2     7 h
## 3     8 i
## 4     9 j
## 5    10 k
```

```r
# rows where 5 divides evenly into `a`
A %>% filter(a %% 5 == 0)
```

```
## # A tibble: 3 x 2
##       a b
##   <int> <chr>
## 1     0 a
## 2     5 f
## 3    10 k
```

```r
# just a particular case
A %>%
  filter(b == "c")
```

```
## # A tibble: 1 x 2
##       a b
##   <int> <chr>
## 1     2 c
```

```r
# a vector of cases:
A %>%
  filter(b %in% c("b", "f", "g"))
```

```
## # A tibble: 3 x 2
##       a b
##   <int> <chr>
## 1     1 b
## 2     5 f
## 3     6 g
```

As you can see, logical evaluation is the key to making intelligent use of `filter()`. You can query columns in the data directly within the filter call. The key is to produce a value of either `TRUE` or `FALSE` for each row of the data. Where the logical expression evaluates to `TRUE` we keep the rows, and `FALSE`s are discarded. Some useful logical operators include 1. `==` test equality 2. `>=` (`<=`) test inclusive greater than (less than) 3. `%in%` test membership 4. `any()` is any element in a vector `TRUE` 5. `all()` are all elements of a vector `TRUE` 6. `!` negation of any of the above 7. `between()` tests if a value is in an interval 8. `&` logical AND 9. `|` logical OR

More examples:

```
A %>%
  # between() is by default inclusive in its bounds
  filter(between(a, 3, 5) | b == "g")
```

```
## # A tibble: 4 x 2
##       a b
##   <int> <chr>
## 1     3 d
## 2     4 e
## 3     5 f
## 4     6 g
```

```
A %>%
  # multiple conditions
  filter(a < 7,
         a >= 2,
         b %in% c("a","c","e","g","i","k"))
```

```
## # A tibble: 3 x 2
##       a b
##   <int> <chr>
## 1     2 c
## 2     4 e
## 3     6 g
```

Note `filter()` accepts comma-separated arguments, interpreting the commas as `&`.

1.5.1 Mini Exercises for filters

1. How many rows of `gapminder` have a life expectancy between 50 and 60, inclusive
2. Which countries have ever had a life expectancy greater than 78?

1.6 selecting is for columns

Sometime we don't need all the columns in the data. We can select particular ones by name or position, like so:

```
gapminder %>%
  select(country, year, gdpPercap)
```

```
## # A tibble: 1,704 x 3
##    country      year gdpPercap
##    <fct>       <int>     <dbl>
##  1 Afghanistan  1952      779.
##  2 Afghanistan  1957      821.
##  3 Afghanistan  1962      853.
```

```
##   4 Afghanistan  1967      836.
##   5 Afghanistan  1972      740.
##   6 Afghanistan  1977      786.
##   7 Afghanistan  1982      978.
##   8 Afghanistan  1987      852.
##   9 Afghanistan  1992      649.
##  10 Afghanistan  1997      635.
## # ... with 1,694 more rows
## # i Use `print(n = ...)` to see more rows
```

```
gapminder %>%
  select(1,2,6)
```

```
## # A tibble: 1,704 x 3
##    country     continent gdpPercap
##    <fct>       <fct>         <dbl>
##  1 Afghanistan Asia           779.
##  2 Afghanistan Asia           821.
##  3 Afghanistan Asia           853.
##  4 Afghanistan Asia           836.
##  5 Afghanistan Asia           740.
##  6 Afghanistan Asia           786.
##  7 Afghanistan Asia           978.
##  8 Afghanistan Asia           852.
##  9 Afghanistan Asia           649.
## 10 Afghanistan Asia           635.
## # ... with 1,694 more rows
## # i Use `print(n = ...)` to see more rows
```

You could also select columns based on various kinds of column name string matching, like so

```
gapminder %>%
  select(contains("p"))
```

```
## # A tibble: 1,704 x 3
##    lifeExp      pop gdpPercap
##      <dbl>    <int>     <dbl>
##  1    28.8  8425333      779.
##  2    30.3  9240934      821.
##  3    32.0 10267083      853.
##  4    34.0 11537966      836.
##  5    36.1 13079460      740.
##  6    38.4 14880372      786.
##  7    39.9 12881816      978.
##  8    40.8 13867957      852.
##  9    41.7 16317921      649.
## 10    41.8 22227415      635.
## # ... with 1,694 more rows
## # i Use `print(n = ...)` to see more rows
```

```
# or
gapminder %>%
  select(ends_with("p"))
```

```
## # A tibble: 1,704 x 3
```

```
##    lifeExp      pop gdpPercap
##      <dbl>    <int>     <dbl>
##  1    28.8  8425333      779.
##  2    30.3  9240934      821.
##  3    32.0 10267083      853.
##  4    34.0 11537966      836.
##  5    36.1 13079460      740.
##  6    38.4 14880372      786.
##  7    39.9 12881816      978.
##  8    40.8 13867957      852.
##  9    41.7 16317921      649.
## 10    41.8 22227415      635.
## # ... with 1,694 more rows
## # i Use `print(n = ...)` to see more rows
```

Some other useful options are `starts_with()`, and so on. The functions `contains()`, `starts_with()` and `ends_with()` are selecting columns using logical expressions, just as we've done for filtering rows. Try typing `?starts_with` into the console, and pick the help file from the `tidyselect` package to see a listing of other helper functions for column selection.

1.7 create columns with `mutate()`

To create a column, potentially using other columns you already have, use `mutate()` like so:

```
gapminder %>%
  mutate(GDP = pop * gdpPercap)
```

```
## # A tibble: 1,704 x 7
##    country     continent  year lifeExp      pop gdpPercap           GDP
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>         <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.   6567086330.
##  2 Afghanistan Asia       1957    30.3  9240934      821.   7585448670.
##  3 Afghanistan Asia       1962    32.0 10267083      853.   8758855797.
##  4 Afghanistan Asia       1967    34.0 11537966      836.   9648014150.
##  5 Afghanistan Asia       1972    36.1 13079460      740.   9678553274.
##  6 Afghanistan Asia       1977    38.4 14880372      786.  11697659231.
##  7 Afghanistan Asia       1982    39.9 12881816      978.  12598563401.
##  8 Afghanistan Asia       1987    40.8 13867957      852.  11820990309.
##  9 Afghanistan Asia       1992    41.7 16317921      649.  10595901589.
## 10 Afghanistan Asia       1997    41.8 22227415      635.  14121995875.
## # ... with 1,694 more rows
## # i Use `print(n = ...)` to see more rows
```

You can create several columns at once, by separating with commas, and columns can be sequentially dependent. Observe:

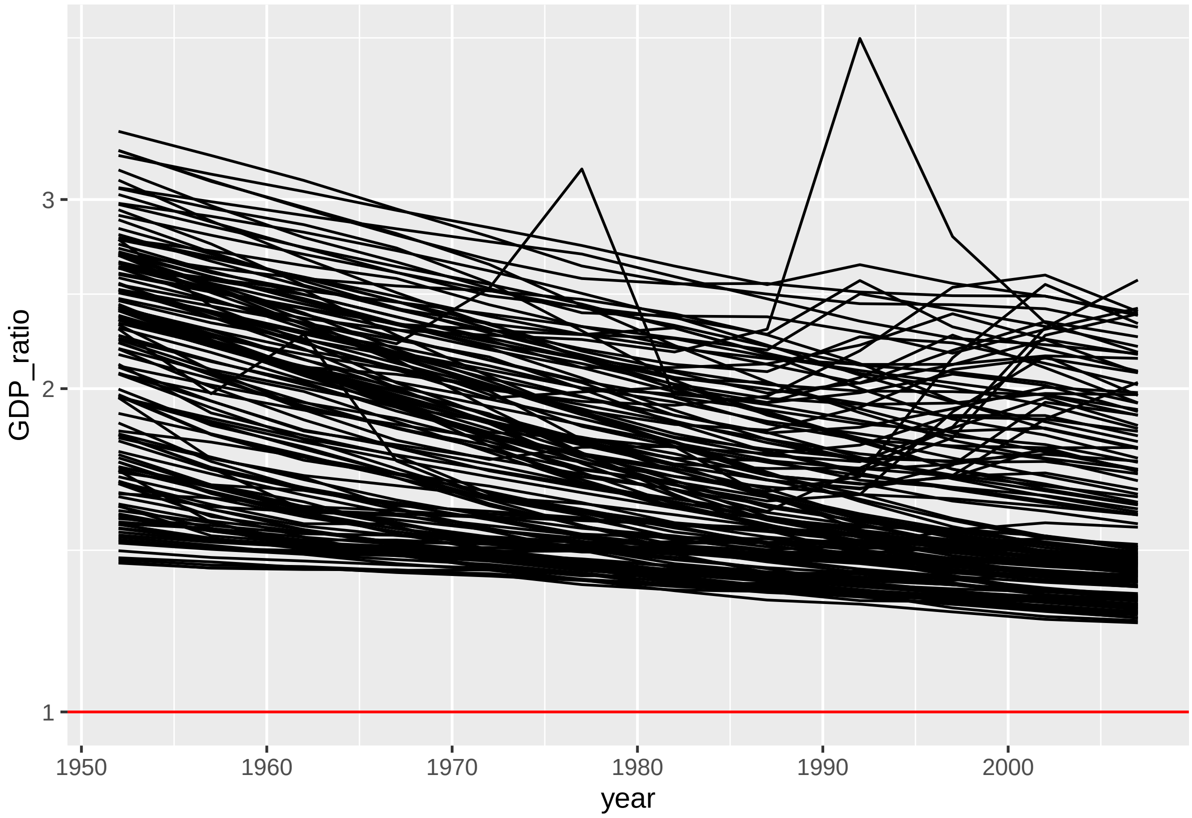
```
gapminder_hypothetical <-
  gapminder %>%
  mutate(GDP = pop * gdpPercap,
         GDP100 = gdpPercap * 100,
         stationary_births = pop / lifeExp,
         GDP_alternative = stationary_births * GDP100,
         GDP_ratio = GDP_alternative / GDP)
```

Note that `GDP`, `GDP100`, `stationary_births`, and `GDP_alternative` are created, and each used later as well within the same `mutate()` call. Whenever it makes sense in your calculations, it's nice to include such operations in a single `mutate()` call rather than in a series of `mutate()` calls. It's cleaner that way. These calculations are meant to showcase `mutate()` usability, not

Aside: What are those strange measures? They're purely hypothetical and likely not useful extrapolations of the data points give, making strange invocations of a stationary world. `GDP` is a direct calculation, not controversial. `GDP100` is calculated on the assumption that if a newborn were to accumulate on average `gdpPercap` of income per year over 100 years, how much would the lifetime `gdpPercap` be? `stationary_births` makes use of the stationary relationship $b = N/e(0)$, meaning that the stationary crude birth rate b is the population size `N` divided by life expectancy. For us `N` is `pop`, but of course neither the population size/structure nor period life expectancy are actually stationary. So this quantity is not usable per se. We then create `GDP_alternative`, which scales `GDP100` to a hypothetical cohort size and says "what would be the lifetime GDP in this fake longevous stationary population?". Finally, we take the ratio. It seems lengthening life to a consistent 100 years would (under these unrealistic constraints) increase GDP.

What does the result look like? (plot code discussed later in workshop)

```
gapminder_hypothetical %>%
  ggplot(aes(x = year, y = GDP_ratio, group = country)) +
  geom_line() +
  scale_y_log10() +
  geom_hline(yintercept = 1, color = "red")
```



Remember that since `mutate()` just creates columns, it does not change the number of rows in the data.

1.7.1 Mini exercise

1. Take 2 minutes to write down as many problems as you can think of in the above analysis, no coding required.

1.8 Create aggregate measures using summarize()

You can create aggregate measures using `summarize()`, implying a likely reduction in the number of rows in the data. For example, we can calculate the total GDP for the most recent year in the data as follows:

```
gapminder %>%
  filter(year == max(year)) %>%
  summarize(GDP = sum(pop * gdpPercap))
```

```
## # A tibble: 1 x 1
##        GDP
##      <dbl>
## 1 5.81e13
```

1.8.1 Mini Exercise for summarize()

1. Calculate the average life expectancy over all countries in the most recent year.
2. Calculate the population weighted average of life expectancy in the most recent year.

The formula for a weighted average is:

$$\bar{x} = \frac{\sum x_i \cdot w_i}{\sum w_i}$$

where x is the thing being weighted and w are the weights. We use this formula in many many places in demography!

```
#
```

1.9 Use group_by() to scale up!

The value of `mutate()` and `summarize()` increases greatly if we learn to make intelligent use of these constructs for subgroups in the data. What if we want to pick out the highest life expectancy per year in the data? To get the highest life expectancy in the data we do:

```
gapminder %>%
  filter(lifeExp == max(lifeExp))
```

```
## # A tibble: 1 x 6
##   country continent  year lifeExp       pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>     <int>     <dbl>
## 1 Japan   Asia       2007    82.6 127467972    31656.
```

To do this for each year, use `group_by()` to impose independent groups in the data on the basis of variables, then do everything else just the same:

```
gapminder %>%
  # apply independent groups
  group_by(year) %>%
  # whatever we do here is independent within groups!
  filter(lifeExp == max(lifeExp)) %>%
```

```r
  # remove when done
  ungroup() %>%
  # sort
  arrange(year)
```

```
## # A tibble: 12 x 6
##    country continent  year lifeExp       pop gdpPercap
##    <fct>   <fct>     <int>   <dbl>     <int>     <dbl>
##  1 Norway  Europe     1952    72.7   3327728    10095.
##  2 Iceland Europe     1957    73.5    165110     9244.
##  3 Iceland Europe     1962    73.7    182053    10350.
##  4 Sweden  Europe     1967    74.2   7867931    15258.
##  5 Sweden  Europe     1972    74.7   8122293    17832.
##  6 Iceland Europe     1977    76.1    221823    19655.
##  7 Japan   Asia       1982    77.1 118454974    19384.
##  8 Japan   Asia       1987    78.7 122091325    22376.
##  9 Japan   Asia       1992    79.4 124329269    26825.
## 10 Japan   Asia       1997    80.7 125956499    28817.
## 11 Japan   Asia       2002    82   127065841    28605.
## 12 Japan   Asia       2007    82.6 127467972    31656.
```

Note, we should remove groups with `ungroup()` when we're done using them, and we sort the data on year for visual inspection using `arrange()`. This reads as "first take `gapminder`, then `group_by()` year, then `filter()` out the highest life expectancy per year, then remove the groups and sort years". Notice how the functions can be read as verbs, and the the pipes allow them to be combined into a rote kind of sentence. Indeed, it can help to add notes using `#`: don't worry: it won't break the chain! As the dataset goes down the pipeline, by default it becomes the first argument to the next function to be executed. Each of these functions has a first argument called either `data` or `.data`, which doesn't need to be specified because the incoming data is passed to it.

Note: you can run this code by simply placing the cursor anywhere in the pipeline and pressing `Ctrl + Enter`. There is no need to select the whole statement before running, although this also works (you could in this case also click the green play arrow).

1.9.1 Exercises for `group_by()`

1. Aggregate all variables by continent and year. For `lifeExp` and `gdpPercap`, use the population weighted averages, as we did above. Is this a job for `mutate()` or `summarize()`?
2. Calculate year-on-year life expectancy changes for each country. Tip, use `lead()` or `lag()`, like so:

```r
# a series incrementing in steps of 1.
a <- c(1,3,4,5,7,10)
lead(a) - a
```

```
## [1]  2  1  1  2  3 NA
```

```r
# *or*
a - lag(a)
```

```
## [1] NA  2  1  1  2  3
```

Note, the first or last year will get `NA`s. You should ensure that years are sorted within countries using `arrange(country, year)`. Is this a job for `mutate()` or `summarize()`? I personally would use the lead() version of this approach.

1.10 reshape using pivot_wider() and pivot_longer()

The gapminder data is already tidy, and most often we use long / wide reshaping operations to force non-tidy data into a tidy form. A wide version of gapminder might, for example, have years spread over columns rather than stacked. Sometimes government statistical offices distribute data like so. Here names_from is the column whose values will determine the new column names, whereas values_from is where data will be drawn from.

```
gapminder_wide <-
  gapminder %>%
  select(country, year, continent, pop) %>%
  pivot_wider(names_from = year,
              values_from = pop)
```

At times, we actually want our data to look like this for some ad hoc calculation convenience. Note, you can list more than one values_from column. We could spread each variable-year combinations by specifying them in a vector. In this case, newly created column names will be concatenated.

```
gapminder %>%
  pivot_wider(names_from = year,
              values_from = c(lifeExp, gdpPercap, pop))
```

```
## # A tibble: 142 x 38
##    country     continent lifeE~1 lifeE~2 lifeE~3 lifeE~4 lifeE~5 lifeE~6 lifeE~7
##    <fct>       <fct>       <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
##  1 Afghanistan Asia         28.8    30.3    32.0    34.0    36.1    38.4    39.9
##  2 Albania     Europe       55.2    59.3    64.8    66.2    67.7    68.9    70.4
##  3 Algeria     Africa       43.1    45.7    48.3    51.4    54.5    58.0    61.4
##  4 Angola      Africa       30.0    32.0    34      36.0    37.9    39.5    39.9
##  5 Argentina   Americas     62.5    64.4    65.1    65.6    67.1    68.5    69.9
##  6 Australia   Oceania      69.1    70.3    70.9    71.1    71.9    73.5    74.7
##  7 Austria     Europe       66.8    67.5    69.5    70.1    70.6    72.2    73.2
##  8 Bahrain     Asia         50.9    53.8    56.9    59.9    63.3    65.6    69.1
##  9 Bangladesh  Asia         37.5    39.3    41.2    43.5    45.3    46.9    50.0
## 10 Belgium     Europe       68      69.2    70.2    70.9    71.4    72.8    73.9
## # ... with 132 more rows, 29 more variables: lifeExp_1987 <dbl>,
## #   lifeExp_1992 <dbl>, lifeExp_1997 <dbl>, lifeExp_2002 <dbl>,
## #   lifeExp_2007 <dbl>, gdpPercap_1952 <dbl>, gdpPercap_1957 <dbl>,
## #   gdpPercap_1962 <dbl>, gdpPercap_1967 <dbl>, gdpPercap_1972 <dbl>,
## #   gdpPercap_1977 <dbl>, gdpPercap_1982 <dbl>, gdpPercap_1987 <dbl>,
## #   gdpPercap_1992 <dbl>, gdpPercap_1997 <dbl>, gdpPercap_2002 <dbl>,
## #   gdpPercap_2007 <dbl>, pop_1952 <int>, pop_1957 <int>, pop_1962 <int>, ...
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

To go in the other direction we use pivot_longer(). This is the more commonly used of the two in practice. Here we specify a column range, then give the names as strings for the columns that will collect the header names and the values. In this case, we give the columns by name, but since they're numbers, we put the names in back tics to ensure they will be interpreted as names rather than as positions!

```
gapminder_wide %>%
  pivot_longer(`1952`:`2007`,
```

```
               names_to = "year",
               values_to = "pop")
```

```
## # A tibble: 1,704 x 4
##    country     continent year       pop
##    <fct>       <fct>     <chr>    <int>
##  1 Afghanistan Asia      1952   8425333
##  2 Afghanistan Asia      1957   9240934
##  3 Afghanistan Asia      1962  10267083
##  4 Afghanistan Asia      1967  11537966
##  5 Afghanistan Asia      1972  13079460
##  6 Afghanistan Asia      1977  14880372
##  7 Afghanistan Asia      1982  12881816
##  8 Afghanistan Asia      1987  13867957
##  9 Afghanistan Asia      1992  16317921
## 10 Afghanistan Asia      1997  22227415
## # ... with 1,694 more rows
## # i Use `print(n = ...)` to see more rows
```

That's fine if you happen to know the column names and if they happen to be in a continuous range, as we have here. You could also use position like so (if you know them)

```
gapminder_wide %>%
  pivot_longer(3:14,
              names_to = "year",
              values_to = "pop")
```

```
## # A tibble: 1,704 x 4
##    country     continent year       pop
##    <fct>       <fct>     <chr>    <int>
##  1 Afghanistan Asia      1952   8425333
##  2 Afghanistan Asia      1957   9240934
##  3 Afghanistan Asia      1962  10267083
##  4 Afghanistan Asia      1967  11537966
##  5 Afghanistan Asia      1972  13079460
##  6 Afghanistan Asia      1977  14880372
##  7 Afghanistan Asia      1982  12881816
##  8 Afghanistan Asia      1987  13867957
##  9 Afghanistan Asia      1992  16317921
## 10 Afghanistan Asia      1997  22227415
## # ... with 1,694 more rows
## # i Use `print(n = ...)` to see more rows
```

Or we could exploit the numeric nature of the column names and use one of the `tidyselect` functions we saw before.

```
gapminder_wide %>%
  pivot_longer(num_range(prefix = "", range = 1952:2007),
               names_to = "year",
               values_to = "pop")
```

```
## # A tibble: 1,704 x 4
##    country     continent year       pop
##    <fct>       <fct>     <chr>    <int>
```

```
##  1 Afghanistan Asia      1952   8425333
##  2 Afghanistan Asia      1957   9240934
##  3 Afghanistan Asia      1962  10267083
##  4 Afghanistan Asia      1967  11537966
##  5 Afghanistan Asia      1972  13079460
##  6 Afghanistan Asia      1977  14880372
##  7 Afghanistan Asia      1982  12881816
##  8 Afghanistan Asia      1987  13867957
##  9 Afghanistan Asia      1992  16317921
## 10 Afghanistan Asia      1997  22227415
## # ... with 1,694 more rows
## # i Use `print(n = ...)` to see more rows
```

Note that `pivot_longer()` also accepts multiple `names_to` destinations, if you happen to have concatenated names, as in the complex `pivot_wider()` example. Here we make and then undo the ugly case above with concatenated variable and year column names. This needs to be done in two steps: first we store the full column range (`3:ncol(.)`, where `ncol(.)` gives the number of columns of the incoming data), splitting names into two new columns for variables and years, and putting all variables together. At this stage the data is *too* long to be easily usable, and we still need to spread variables over the columns, so the final `pivot_wider()` statement does this for us.

```
gapminder %>%
  pivot_wider(names_from = year,
              values_from = c(lifeExp, pop, gdpPercap)) %>%
  # two steps: 1) collect it all
  pivot_longer(3:ncol(.),
               values_to = "value",
               names_to = c("variable","year"),
               names_sep = "_") %>%
  # 2) spread variables back out
  pivot_wider(names_from = variable,
              values_from = value)
```

```
## # A tibble: 1,704 x 6
##    country     continent year  lifeExp      pop gdpPercap
##    <fct>       <fct>     <chr>   <dbl>    <dbl>     <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.
##  6 Afghanistan Asia      1977     38.4 14880372      786.
##  7 Afghanistan Asia      1982     39.9 12881816      978.
##  8 Afghanistan Asia      1987     40.8 13867957      852.
##  9 Afghanistan Asia      1992     41.7 16317921      649.
## 10 Afghanistan Asia      1997     41.8 22227415      635.
## # ... with 1,694 more rows
## # i Use `print(n = ...)` to see more rows
```

1.10.1 Mini exercises

1. Take the `gapminder` data, remove the `continent` column, then spread `country` over columns using `pivot_wider()`.

2. Undo the result of part 1 of this exercise to return to the original `gapminder` data.

2 Looking ahead

Tomorrow we will do a longer worked example using these basic `tidyverse` operations and introducing some other recoding and dataset joining operations, on the example of fertility data.

Wickham, Hadley. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer.

Wilkinson, Leland. 2012. "The Grammar of Graphics." In *Handbook of Computational Statistics*, 375–414. Springer.