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 **2**
1.1 Definition . . . 2
1.2 Example (gapminder) . . . 2
1.3 Basic dataset descriptives . . . 3
1.3.1 Mini exercise . . . 3
1.4 pipes . . . 4
1.4.1 Mini Exercise for pipes . . . 4
1.5 filtering is for rows . . . 4
1.5.1 Mini Exercises for filters . . . 6
1.6 selecting is for columns . . . 6

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] 0.5729841
```

One can chain together a sequence of operations like so:

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

```
##  [1] 0.2054061 0.4123044 0.7118740 1.0867443 1.5532065 2.2592218 3.0148123
##  [8] 3.7982460 4.6139482 5.4890858
```

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
```

```
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
```

```
# 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
```

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

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

```
# 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
```

```r
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

```r
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
```

```r
# 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.

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.