Data Wrangling and Cleaning

R. S. Verspoor

2024-06-012

It's estimated that data scientists spend between 50-80% of their time cleaning data into a format they can use for analysis. This process is called data wrangling and is important in a world of big data. This workshop will run you through some of the key core functions for cleaning, manipulating and summarising data. We will be using the tidyverse packages tidyr and dplyr that are designed specifically to help with data wrangling.

To get started we will need to install and load `tidyverse`. Note that this will load a suite of tidyverse packages, of which we will use those detailed above.

```
library(tidyverse)
```

Tidy data

Hadley Wickham, who created the tidyverse, distinguishes between two types of data set: tidy and messy. This makes a distinction between a specific way of arranging data to make it useful for most R analyses.

Specifically, a tidy data set is one in which:

- rows contain different observations;
- columns contain different variables;
- cells contain values.

1. Simple manipulations

Let's begin by exploring the iris data set, which gives the measurements in centimeters of the variables sepal length and width, and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa, versicolor*, and *virginica*. This data set is available as part of the base R package. Let's have a look at the data:

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Is this dataset 'tidy'?

◯ Yes

◯ No

Show / Hide result

Let's start by looking at some basic operations, such as subsetting, sorting and adding new columns.

1.1 Filtering rows

One operation we often want to do is to extract a subset of rows according to some criterion. For example, we may want to extract all rows of the `iris` dataset that correspond to the `versicolor` species. In tidyverse, we can use a function called `filter()`:

```
filter(iris, Species == "versicolor")
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1           7.0         3.2          4.7         1.4 versicolor
## 2           6.4         3.2          4.5         1.5 versicolor
## 3           6.9         3.1          4.9         1.5 versicolor
## 4           5.5         2.3          4.0         1.3 versicolor
## 5           6.5         2.8          4.6         1.5 versicolor
## 6           5.7         2.8          4.5         1.3 versicolor
## 7           6.3         3.3          4.7         1.6 versicolor
## 8           4.9         2.4          3.3         1.0 versicolor
## 9           6.6         2.9          4.6         1.3 versicolor
## 10          5.2         2.7          3.9         1.4 versicolor
##  [ reached 'max' / getOption("max.print") -- omitted 40 rows ]
```

The first argument to the `filter()` function is the data, and the second corresponds to the criteria for filtering. Notice that we did not need to use the $ operator in the `filter()` function. As with ggplot2 the `filter()` function knows to look for the column Species in the data set iris.

1.2 Sorting rows

Another common operation is to sort rows according to some criterion. Let's try to sort rows by `Species` and then `Sepal.Length`. In tidyverse we can use the `arrange()` function.

```
arrange(iris, Species, Sepal.Length)
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           4.3         3.0          1.1         0.1  setosa
## 2           4.4         2.9          1.4         0.2  setosa
## 3           4.4         3.0          1.3         0.2  setosa
## 4           4.4         3.2          1.3         0.2  setosa
## 5           4.5         2.3          1.3         0.3  setosa
## 6           4.6         3.1          1.5         0.2  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           4.6         3.6          1.0         0.2  setosa
## 9           4.6         3.2          1.4         0.2  setosa
## 10          4.7         3.2          1.3         0.2  setosa
##  [ reached 'max' / getOption("max.print") -- omitted 140 rows ]
```

Notice once again that the first argument to `arrange()` is the data set, and then subsequent arguments are the columns that we wish to order by. Again, we do not require the $ operator here.

1.3 Selecting columns

Now let's say we want to select just the `Species`, `Sepal.Length` and `Sepal.Width` columns from the data set. In tidyverse we can use the `select()` function.

```
select(iris, Species, Sepal.Length, Sepal.Width)
```

```
##    Species Sepal.Length Sepal.Width
## 1   setosa          5.1         3.5
## 2   setosa          4.9         3.0
## 3   setosa          4.7         3.2
## 4   setosa          4.6         3.1
## 5   setosa          5.0         3.6
## 6   setosa          5.4         3.9
## 7   setosa          4.6         3.4
## 8   setosa          5.0         3.4
## 9   setosa          4.4         2.9
## 10  setosa          4.9         3.1
## 11  setosa          5.4         3.7
## 12  setosa          4.8         3.4
## 13  setosa          4.8         3.0
## 14  setosa          4.3         3.0
## 15  setosa          5.8         4.0
## 16  setosa          5.7         4.4
##  [ reached 'max' / getOption("max.print") -- omitted 134 rows ]
```

Notice once again that the first argument to `select()` is the data set, and then subsequent arguments are the columns that we wish to select; no $ operators required.

There is even a set of functions to help extract columns based on pattern matching e.g.

```
select(iris, Species, starts_with("Sepal"))
```

Note that we can also remove columns using a `-` operator e.g.

```
select(iris, -starts_with("Petal"))
```

or

```
select(iris, -Petal.Length, -Petal.Width)
```

would remove the petal columns.

1.4 Adding columns

Finally, let's add a new column called `Sepal.Length2` that contains the square of the sepal length. In tidyverse this would be:

```
mutate(iris, Sepal.Length2 = Sepal.Length^2)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Length2
## 1          5.1         3.5          1.4         0.2  setosa         26.01
## 2          4.9         3.0          1.4         0.2  setosa         24.01
## 3          4.7         3.2          1.3         0.2  setosa         22.09
## 4          4.6         3.1          1.5         0.2  setosa         21.16
## 5          5.0         3.6          1.4         0.2  setosa         25.00
## 6          5.4         3.9          1.7         0.4  setosa         29.16
## 7          4.6         3.4          1.4         0.3  setosa         21.16
## 8          5.0         3.4          1.5         0.2  setosa         25.00
##  [ reached 'max' / getOption("max.print") -- omitted 142 rows ]
```

1.5 Pipes

Piping comes from Unix scripting, and simply means a chain of commands, such that the results from each command feed into the next one.Recently, the `magrittr` package, and subsequently `tidyverse` have introduced the pipe operator `%>%` that enables us to chain functions together. Let's look at an example:

```
iris %>% filter(Species == "versicolor")
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1           7.0         3.2          4.7         1.4 versicolor
## 2           6.4         3.2          4.5         1.5 versicolor
## 3           6.9         3.1          4.9         1.5 versicolor
## 4           5.5         2.3          4.0         1.3 versicolor
## 5           6.5         2.8          4.6         1.5 versicolor
## 6           5.7         2.8          4.5         1.3 versicolor
## 7           6.3         3.3          4.7         1.6 versicolor
## 8           4.9         2.4          3.3         1.0 versicolor
## 9           6.6         2.9          4.6         1.3 versicolor
## 10          5.2         2.7          3.9         1.4 versicolor
##  [ reached 'max' / getOption("max.print") -- omitted 40 rows ]
```

Notice: when we did this before we would write something like `filter(iris, Species == "versicolor")` i.e. we required the first argument of `filter()` to be a data.frame (or tibble). The pipe operator `%>%` does this automatically, so the outcome from the left-hand side of the operator is passed as the first argument to the right-hand side function. This makes the code more succinct, and easier to read (because we are not repeating pieces of code).

Pipes can be chained together multiple times. For example:

```
iris %>%
    filter(Species == "versicolor") %>%
    select(Species, starts_with("Sepal")) %>%
    mutate(Sepal.Length2 = Sepal.Length^2) %>%
    arrange(Sepal.Length)
```

```
##       Species Sepal.Length Sepal.Width Sepal.Length2
## 1  versicolor          4.9         2.4         24.01
## 2  versicolor          5.0         2.0         25.00
## 3  versicolor          5.0         2.3         25.00
## 4  versicolor          5.1         2.5         26.01
## 5  versicolor          5.2         2.7         27.04
## 6  versicolor          5.4         3.0         29.16
## 7  versicolor          5.5         2.3         30.25
## 8  versicolor          5.5         2.4         30.25
## 9  versicolor          5.5         2.4         30.25
## 10 versicolor          5.5         2.5         30.25
## 11 versicolor          5.5         2.6         30.25
## 12 versicolor          5.6         2.9         31.36
##  [ reached 'max' / getOption("max.print") -- omitted 38 rows ]
```

Notice that the pipe operator must be at the end of the line if we wish to split the code over multiple lines.

In essence we can read what we have done in much the same way as if we were reading prose. Firstly we take the `iris` data, `filter` to extract just those rows corresponding to `versicolor` species, `select` species and sepal measurements, `mutate` the data frame to contain a new column that is the square of the sepal lengths and finally `arrange` in order of increasing sepal length.

Once we've got our head around pipes, we can begin to use some of the other useful functions in tidyverse to do some really useful things.

2. Grouping and summarising

A common thing we might want to do is to produce summaries of some variable for different subsets of the data. For example, we might want to produce an estimate of the mean of the sepal lengths for each species of iris. The `dplyr` package provides a `function group_by()` that allows us to group data, and `summarise()` that allows us to summarise data.

In this case we can think of what we want to do as “grouping” the data by `Species` and then averaging the `Sepal.Length` values within each group. Hence,

```
iris %>%
    group_by(Species) %>%
    summarise(mn = mean(Sepal.Length))
```

```
## # A tibble: 3 × 2
##   Species       mn
##   <fct>      <dbl>
## 1 setosa      5.01
## 2 versicolor  5.94
## 3 virginica   6.59
```

The `summarise()` function (**note**, this is different to the summary() function), applies a function to a `data.frame` or subsets of a `data.frame` .

TASK

Produce a table of estimates for the mean and variance of both sepal lengths and widths, within each species.

3. Reshaping data sets

Another key feature of tidyverse is the power it gives you to reshape data sets. The two key functions are `gather()` and `spread()` . The `gather()` function takes multiple columns, and gathers them into key-value pairs. The `spread()` function is its converse, it takes two columns (key and value) and spreads these into multiple columns. These ideas are best illustrated by an example.

3.1 Example

We will be using the Gapminder GDP per capita data that is found in the `gapminder` package, as well as online: https://www.gapminder.org/data/ (https://www.gapminder.org/data/). Download the csv file called ‘indicator gapminder gdp_per_capita_ppp.csv’ and read the data in using the `read_csv()` function in `tidyverse` .

```
gp_income <- read_csv("indicator gapminder gdp_per_capita_ppp.csv")
gp_income
```

```
## # A tibble: 262 × 217
##    `GDP per capita`      `1800` `1801` `1802` `1803` `1804` `1805` `1806` `1807`
##    <chr>                  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 Abkhazia                  NA     NA     NA     NA     NA     NA     NA     NA
##  2 Afghanistan              603    603    603    603    603    603    603    603
##  3 Akrotiri and Dhekelia     NA     NA     NA     NA     NA     NA     NA     NA
##  4 Albania                  667    667    668    668    668    668    668    668
##  5 Algeria                  716    716    717    718    719    720    721    722
##  6 American Samoa            NA     NA     NA     NA     NA     NA     NA     NA
##  7 Andorra                 1197   1199   1201   1204   1206   1208   1210   1212
##  8 Angola                   618    620    623    626    628    631    634    637
##  9 Anguilla                  NA     NA     NA     NA     NA     NA     NA     NA
## 10 Antigua and Barbuda      757    757    757    757    757    757    757    758
## # ℹ 252 more rows
## # ℹ 208 more variables: `1808` <dbl>, `1809` <dbl>, `1810` <dbl>, `1811` <dbl>,
## #   `1812` <dbl>, `1813` <dbl>, `1814` <dbl>, `1815` <dbl>, `1816` <dbl>,
## #   `1817` <dbl>, `1818` <dbl>, `1819` <dbl>, `1820` <dbl>, `1821` <dbl>,
## #   `1822` <dbl>, `1823` <dbl>, `1824` <dbl>, `1825` <dbl>, `1826` <dbl>,
## #   `1827` <dbl>, `1828` <dbl>, `1829` <dbl>, `1830` <dbl>, `1831` <dbl>,
## #   `1832` <dbl>, `1833` <dbl>, `1834` <dbl>, `1835` <dbl>, `1836` <dbl>, …
```

Is this data in a ‘tidy’ format?

◯ Yes

◯ No

Show / Hide result

Before we go any further, notice that the first column is labelled incorrectly as GDP per capita (this is an artefact from the original data set), so let’s rename the first column using the `rename()` function:

```
gp_income <- gp_income %>%
    rename(country = "GDP per capita")
gp_income
```

```
## # A tibble: 262 × 217
##    country `1800` `1801` `1802` `1803` `1804` `1805` `1806` `1807` `1808` `1809`
##    <chr>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 Abkhaz…     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
##  2 Afghan…    603    603    603    603    603    603    603    603    603    603
##  3 Akroti…     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
##  4 Albania    667    667    668    668    668    668    668    668    668    668
##  5 Algeria    716    716    717    718    719    720    721    722    723    724
##  6 Americ…     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
##  7 Andorra   1197   1199   1201   1204   1206   1208   1210   1212   1215   1217
##  8 Angola     618    620    623    626    628    631    634    637    640    642
##  9 Anguil…     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
## 10 Antigu…    757    757    757    757    757    757    757    758    758    758
## # i 252 more rows
## # i 206 more variables: `1810` <dbl>, `1811` <dbl>, `1812` <dbl>, `1813` <dbl>,
## #   `1814` <dbl>, `1815` <dbl>, `1816` <dbl>, `1817` <dbl>, `1818` <dbl>,
## #   `1819` <dbl>, `1820` <dbl>, `1821` <dbl>, `1822` <dbl>, `1823` <dbl>,
## #   `1824` <dbl>, `1825` <dbl>, `1826` <dbl>, `1827` <dbl>, `1828` <dbl>,
## #   `1829` <dbl>, `1830` <dbl>, `1831` <dbl>, `1832` <dbl>, `1833` <dbl>,
## #   `1834` <dbl>, `1835` <dbl>, `1836` <dbl>, `1837` <dbl>, `1838` <dbl>, …
```

Notice that the `rename()` function takes the same form as other tidyverse functions such as `filter()` or `arrange()`. We then overwrite the original data frame to keep our workspace neat. *Note*: this is OK here because we have a copy of our raw data saved in an external file. This, combined with the use of scripts, means we have a backup of the original data in case anything goes wrong. Don't overwrite your original data set!

The next thing we need to do is to collapse the year columns down. Ideally we want a column corresponding to country, a column corresponding to year and a final column corresponding to GDP. We are going to do this by using the `gather()` function. Note that the arguments to `gather()` are:

- `data`: this gives the name of the data frame;
- `key`: gives the name of the column that will contain the collapsed column names (e.g. `1800`, `1801` etc.);
- `value`: gives the name of the columns that will contain the values in each of the cells of the collapsed column (e.g. the corresponding GDP values);
- the final set of arguments correspond to those columns we wish to collapse. Here we want to collapse everything except `country`, which we can do using the `-` operator.

```
gp_income <- gp_income %>%
    gather(key = year, value = gdp, -country)
gp_income
```

```
## # A tibble: 56,592 × 3
##    country               year    gdp
##    <chr>                 <chr> <dbl>
##  1 Abkhazia              1800     NA
##  2 Afghanistan           1800    603
##  3 Akrotiri and Dhekelia 1800     NA
##  4 Albania               1800    667
##  5 Algeria               1800    716
##  6 American Samoa        1800     NA
##  7 Andorra               1800   1197
##  8 Angola                1800    618
##  9 Anguilla              1800     NA
## 10 Antigua and Barbuda   1800    757
## # i 56,582 more rows
```

This is almost there now. Notice that R has left the new `year` column as a `character` vector, so we want to change that:

```
gp_income <- gp_income %>%
        mutate(year = as.numeric(year))
```

Also, there is quite a lot of extraneous information in the data. Firstly, there were some mostly empty rows in Excel, which manifest as missing values when the data were read into R:

```
sum(is.na(gp_income$country))  total number of missing (NA) values in the country column of the gp_income data frame.
```

```
## [1] 432
```

We can examine these rows by filtering:

```
gp_income %>% filter(is.na(country)) %>% summary()
```

the summary statistics for all numeric columns and non-numeric columns, but only for rows where the country column is NA.

```
##    country              year          gdp
##  Length:432         Min.   :1800   Min.   :36327
##  Class :character   1st Qu.:1854   1st Qu.:36327
##  Mode  :character   Median :1908   Median :36327
##                     Mean   :1908   Mean   :36327
##                     3rd Qu.:1961   3rd Qu.:36327
##                     Max.   :2015   Max.   :36327
##                                    NA's   :431
```

We can see from the summary that only one row has any GDP information, and indeed in the original data there was a single additional point that could be found in cell HE263 of the original Excel file. I think this is an artefact of the original data, and as such we will remove it here:

```
gp_income <- gp_income %>% filter(!is.na(country))
```

We can also remove the rows that have no GDP information if we so wish (which are denoted by missing values—NA):

```
gp_income <- gp_income %>% filter(!is.na(gdp))
```

Finally, we will restrict ourselves to looking at the data from 1990 onwards:

```
gp_income <- gp_income %>% filter(year > 1990)
head(gp_income)
```

```
## # A tibble: 6 × 3
##   country              year   gdp
##   <chr>               <dbl> <dbl>
## 1 Afghanistan          1991  1022
## 2 Albania              1991  3081
## 3 Algeria              1991  9748
## 4 Andorra              1991 28029
## 5 Angola               1991  4056
## 6 Antigua and Barbuda  1991 17361
```

```
summary(gp_income)
```

```
##    country              year          gdp
##  Length:5075        Min.   :1991   Min.   :   142
##  Class :character   1st Qu.:1997   1st Qu.:  2809
##  Mode  :character   Median :2003   Median :  8476
##                     Mean   :2003   Mean   : 15743
##                     3rd Qu.:2009   3rd Qu.: 21950
##                     Max.   :2015   Max.   :148374
```

Phew! This took some effort, but we've managed to end up with a fairly clean data set that we can plot, summarise etc.

To do all of the previous data cleaning operation using **pipes**, we can write:

```
gp_income <- read_csv("indicator gapminder gdp_per_capita_ppp.csv") %>%
               rename(country = `GDP per capita`) %>%
               gather(year, gdp, -country) %>%
               mutate(year = as.numeric(year)) %>%
               filter(!is.na(country)) %>%
               filter(!is.na(gdp)) %>%
               filter(year > 1990)
```

Task

1.1 Now we can begin to summarise the data. Can you produce a mean GDP for each country, averaging over years. In this case we can think of "grouping" the data by country and then averaging the GDP values within each group (as we have seen before).

1.2 Now try to produce the mean GDP for each year, averaged across country.

1.3 Load in the file "indicator hiv estimated prevalence% 15-49.csv". This file contains the estimated HIV prevalence in people of age 15–49 in different countries over time. Prevalence is defined here to be the estimated number of people living with HIV per 100 population. Produce a tidy data set called `gp_hiv` using the tools in `tidyverse` that we introduced above. The dataset needs to run from 1991 onwards, and we want to end up with columns `country`, `year` and `prevalence`. [Note that a couple of the years have no values in the data set, and by default R reads these columns in as `character` columns. Hence when you `gather()` the data to create a `prevalence` column, all the numbers will be converted into characters. One way to deal with this is to convert the column back into numbers once you have filtered away all the stuff!]