

Tidyr

Rodrigo Esteves de Lima-Lopes
State University of Campinas
rll307@unicamp.br

Contents

1	Introduction	1
2	Loading the packages	1
3	‘%>%’ (pipe) operator	2
4	Creating a data frame for use	3
5	Some tidyr functions	3
5.1	gather	4
5.2	spread	4
5.3	Separate	5
5.4	Unite	6

1 Introduction

This tutorial is based on the following on-line sources:

- Reshaping Data Using Tidyr
- Reshaping Your Data with tidy
- R for data science
- Sthda
- UC Business Analytics R Programming Guide
- RStudio Education

My objective here is to manipulate data using some basics on **Tidyr** package. Please, refer to the cheatsheet cheatsheet for further commands.

tidyr makes possible to reorganise and tidy data more easily and consistently. This is particular relevant if we think that language data tends to be specially messy.

2 Loading the packages

Usually **tidyr** is already part of our *R* distribution. If not, we will have to install it.

```
library(tidyr)
library(dplyr)
```

3 ‘%>%’ (pipe) operator

The pipe operator was first introduced in the package `magrittr`. It aims at helping us to write less code as we ‘pass’ the previous elements of a command to the next, saving us some more complex *R* syntax. For example this nested command:

```
head(
  arrange(
    summarize(
      group_by(
        filter(mtcars, carb > 1),
        cyl
      ),
      Avg_mpg = mean(mpg)
    ),
    desc(Avg_mpg)
  )
)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 2
##   cyl Avg_mpg
##   <dbl> <dbl>
## 1     4   25.9
## 2     6   19.7
## 3     8   15.1
```

In a single line it would get much harder to read:

```
head(arrange(summarize(group_by(filter(mtcars, carb > 1), cyl), Avg_mpg = mean(mpg)), desc(Avg_mpg)))
```

If I choose not to nest commands, it might get a lot of coding:

```
a <- filter(mtcars, carb > 1)
b <- group_by(a, cyl)
c <- summarise(b, Avg_mpg = mean(mpg))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
d <- arrange(c, desc(Avg_mpg))
head(d)
```

```
## # A tibble: 3 x 2
##   cyl Avg_mpg
##   <dbl> <dbl>
## 1     4   25.9
## 2     6   19.7
## 3     8   15.1
```

So, It would become much simpler:

```
mtcars %>%
  filter(carb > 1) %>%
  group_by(cyl) %>%
  summarise(Avg_mpg = mean(mpg)) %>%
  arrange(desc(Avg_mpg)) %>%
  head()
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 2
##   cyl Avg_mpg
##   <dbl> <dbl>
## 1     4   25.9
## 2     6   19.7
## 3     8   15.1
```

4 Creating a data frame for use

```
n<-10
wide <- data.frame(ID = c(1:n)%>%
  paste0("ID",.),
  cr.1=runif(n, min = 1, max = 25),
  cr.2=runif(n, min = 1, max = 25),
  cr.3=runif(n, min = 1, max = 25))
head(wide)
```

```
##   ID      cr.1      cr.2      cr.3
## 1 ID1  9.788189  6.225421 10.124239
## 2 ID2  7.578921  1.806827  7.271789
## 3 ID3 11.092809 12.966421 19.240177
## 4 ID4 10.238812 11.386429  9.019144
## 5 ID5 13.515602  9.638050 21.831011
## 6 ID6 12.076641 16.263747  6.917716
```

5 Some tidyr functions

In this tutorial we will focus on some of **tidyr** functions:

- **gather()** collapses columns un a key-paired set of values
- **spread()** reverses **gather()**, it makes multiples columns from 1
- **separate()** splits a single column into multiple columns
- **unite()** combines multiple columns into a single column

A visual representation of such actions would be:

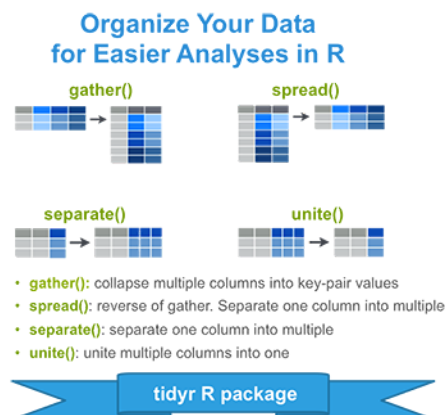


Figure 1: STDHA representation for **tidyr**

5.1 gather

`gather()` helps us to collapse columns into rows. It is common to use this command to gather similar elements within a single column.

```
long <- wide %>%  
  gather(Cr, Freq, cr.1:cr.3)  
head(long)
```

```
##   ID   Cr      Freq  
## 1 ID1 cr.1  9.788189  
## 2 ID2 cr.1  7.578921  
## 3 ID3 cr.1 11.092809  
## 4 ID4 cr.1 10.238812  
## 5 ID5 cr.1 13.515602  
## 6 ID6 cr.1 12.076641
```

Some arguments of `gather()` that we might be using:

- **data**: Your data frame.
- **key, value**: The new names of the columns I will create in the output.
- **...**: The columns to gather. Use the existing variable names.
- **na.rm**: If `rm=TRUE`, it removes NA values

Naturally there are other arguments, but these are the ones we use quite frequently. RStudio Education offers a nice visual representation of `gather()`:

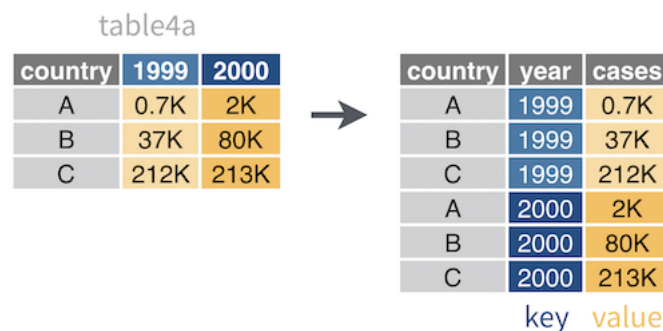


Figure 2: `Gather()`: from RStudio-Education

5.2 spread

`spread()` reshapes a data frame into the wider format. If we take the same data frame we just created and apply this command, we will have our original data frame back:

```
wide.2 <- long %>%  
  spread(Cr, Freq)  
wide.2
```

```
##   ID      cr.1      cr.2      cr.3  
## 1  ID1  9.788189  6.225421 10.124239  
## 2 ID10 14.679058  2.766765 13.648597  
## 3  ID2  7.578921  1.806827  7.271789  
## 4  ID3 11.092809 12.966421 19.240177  
## 5  ID4 10.238812 11.386429  9.019144
```

```
## 6    ID5 13.515602  9.638050 21.831011
## 7    ID6 12.076641 16.263747  6.917716
## 8    ID7 20.198403 16.765303 18.052205
## 9    ID8 10.653752  4.760038 16.980659
## 10   ID9 15.307884 14.155540  5.045495
```

The main arguments of `spread()` are:

key: The name of the column where the headings are. **value:** The values that will populate the rows.

A good visual representation of `spread()` would be:

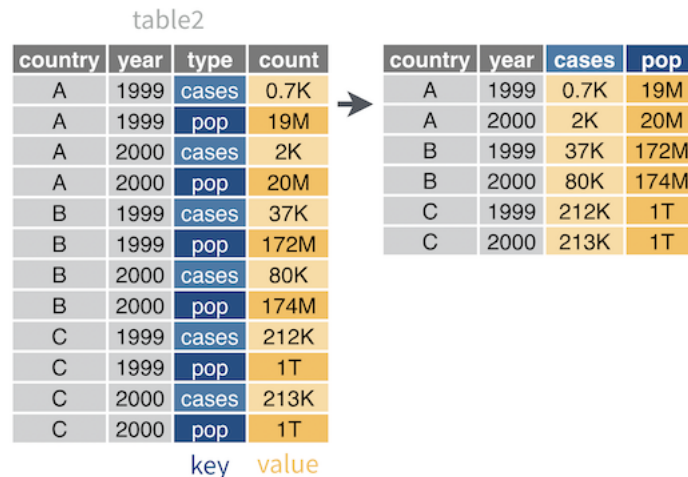


Figure 3: `Spread()` from RStudio-Education

5.3 Separate

`separate()` breaks expressions in a same column using a character as basis. For example, in our dataset:

```
head(long,10)
```

```
##      ID   Cr      Freq
## 1   ID1 cr.1  9.788189
## 2   ID2 cr.1  7.578921
## 3   ID3 cr.1 11.092809
## 4   ID4 cr.1 10.238812
## 5   ID5 cr.1 13.515602
## 6   ID6 cr.1 12.076641
## 7   ID7 cr.1 20.198403
## 8   ID8 cr.1 10.653752
## 9   ID9 cr.1 15.307884
## 10 ID10 cr.1 14.679058
```

`Cr` column brings data in the format `cr+.+number`. `Separate` would break `Cr` into two columns:

```
long_separate <- long %>%
  separate(Cr, c("Feature", "Number"))
head(long_separate,10)
```

```
##      ID Feature Number      Freq
```

```
## 1 ID1 cr 1 9.788189
## 2 ID2 cr 1 7.578921
## 3 ID3 cr 1 11.092809
## 4 ID4 cr 1 10.238812
## 5 ID5 cr 1 13.515602
## 6 ID6 cr 1 12.076641
## 7 ID7 cr 1 20.198403
## 8 ID8 cr 1 10.653752
## 9 ID9 cr 1 15.307884
## 10 ID10 cr 1 14.679058
```

The main arguments of `separate` are:

- `col`: column to be broken into others
- `into`: names for the new column
- `remove`: logical, if true deletes the input column

A good visual representation of `separate()` would be:

table3

country	year	rate		country	year	cases	pop
A	1999	0.7K/19M		A	1999	0.7K	19M
A	2000	2K/20M		A	2000	2K	20M
B	1999	37K/172M		B	1999	37K	172
B	2000	80K/174M		B	2000	80K	174
C	1999	212K/1T		C	1999	212K	1T
C	2000	213K/1T		C	2000	213K	1T

Figure 4: `Separate()` from RStudio-Education

5.4 Unite

`unite()` does exactly the other way around: it unites some columns in a single one:

```
long_unite <- long_separate %>%
  unite(Cr, Feature, Number, sep = "/")
head(long_unite,10)
```

```
## ID Cr Freq
## 1 ID1 cr/1 9.788189
## 2 ID2 cr/1 7.578921
## 3 ID3 cr/1 11.092809
## 4 ID4 cr/1 10.238812
## 5 ID5 cr/1 13.515602
## 6 ID6 cr/1 12.076641
## 7 ID7 cr/1 20.198403
## 8 ID8 cr/1 10.653752
## 9 ID9 cr/1 15.307884
## 10 ID10 cr/1 14.679058
```

The only difference is that now I have chosen to use a "/" as a separator. The main arguments of `unite` are:


- `Data`: My data frame
- `col`: the name of the new column
- `...`: Columns to unite

- **sep**: Separator to use between values

A good visual representation of `unite()` would be:

table5

country	century	year
Afghan	19	99
Afghan	20	0
Brazil	19	99
Brazil	20	0
China	19	99
China	20	0



country	year
Afghan	1999
Afghan	2000
Brazil	1999
Brazil	2000
China	1999
China	2000

Figure 5: `Unite()` from RStudio-Education

Please note that no separator was informed in the code above