An Introduction to the Tidyverse

Chapter 3 (1)

Guani Wu

Stats 102A: Introduction to Computational Statistics with R

Acknowledgements: Miles Chen and Michael Tsiang

(This chapter is largely drawn from Hadley Wickham's Advanced R, 2019)

Section 1

Introduction

The Tidyverse

The **`tidyverse`** is a system of R packages designed by Hadley Wickham (creator of `ggplot2` and current Chief Scientist at RStudio), to improve data management, exploration, and visualization in R.

The Core Tidyverse

There are many packages in the tidyverse, but the core ones are:

- `tibble`: A package to introduce a more efficient data frame.
- `readr`: Functions to read rectangular data.
- `tidyr`: Functions to help make data "tidy."
- `dplyr`: Functions for data manipulation.
- `purrr`: Functions to enhance R's functional programming (i.e., vectorization).
- `stringr`: Functions for string (character) manipulation.
- `forcats`: Functions for factors (categorical variables).
- `ggplot2`: A graphics system based on the "Grammar of Graphics."

Tidyverse Resources

We will introduce the most common functions and packages in the tidyverse.
For more information and resources:

- Garrett Grolemund and Hadley Wickham's "R for Data Science": http://r4ds.had.co.nz/
- https://www.rstudio.com/resources/cheatsheets/

Installing the Tidyverse

The easiest way to download the core tidyverse is to install the `tidyverse` package.

```
install.packages("tidyverse")
```

Then you can load the tidyverse.

```
library(tidyverse)
```

The Pipe

All the packages in the tidyverse (except `ggplot2`) were written to be compatible with the **pipe** operator, denoted by %>%, often pronounced as "**and then**". The pipe is meant to help make code easier to read and understand.

The primary way the %>% operator works is to put the object on the lefthand side into the first argument of the function on the righthand side:

`f(x)` can be written as `x %>% f()`.

`f(x,y)` can be written as `x %>% f(y)`.

Side Note: The pipe operator is from the `magrittr` package, but packages in the tidyverse will load %>% automatically.

The Pipe

By extension, we can include multiple pipes:

`x %>% f(y) %>% g(z)` is equivalent to `g(f(x,y),z)`.

When using multiple pipes in one expression, it is more readable to write the pipes on separate lines:

```
x %>%
  f(y) %>%
  g(z)
```

Pipe Examples

```r
pi %>% cos()
```

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

```r
x <- c(1,2,NA,4,5)
x %>% mean(na.rm=TRUE)
```

```
## [1] 3
```

```r
pi %>%
  sin() %>%
  cos()
```

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

Argument Placeholder

The pipe operator can also be used as an argument placeholder by using a `.` on the righthand side to represent the object on the lefthand side.

`f(x,y)` can be written as `y %>% f(x,.)`.

```
trees %>% lm(Volume ~ Height,data=.)
```

```
##
## Call:
## lm(formula = Volume ~ Height, data = .)
##
## Coefficients:
## (Intercept)       Height
##     -87.124        1.543
```

When Not to Use the Pipe

The pipe is a popular and useful tool in R, but it is not the only tool. Pipes are most useful for writing (or rewriting) a short linear sequence of operations.

Consider using other approaches if:

- Your pipes too long (e.g., more than ten steps). It is easier to have intermediate objects for longer sequences of operations for interpretability and debugging.
- You have multiple inputs or outputs. The pipe is meant for manipulating one primary object, not for working with two or more objects together.

Section 2

Tibbles

Tibble Example

We will consider the `diamonds` data in the `ggplot2` package.

```r
data(diamonds)
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut   color clarity depth table price
##    <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int>
##  1  0.23 Ideal E     SI2      61.5    55   326
##  2  0.21 Prem~ E     SI1      59.8    61   326
##  3  0.23 Good  E     VS1      56.9    65   327
##  4  0.29 Prem~ I     VS2      62.4    58   334
##  5  0.31 Good  J     SI2      63.3    58   335
##  6  0.24 Very~ J     VVS2     62.8    57   336
##  7  0.24 Very~ I     VVS1     62.3    57   336
##  8  0.26 Very~ H     SI1      61.9    55   337
##  9  0.22 Fair  E     VS2      65.1    61   337
## 10  0.23 Very~ H     VS1      59.4    61   338
## # ... with 53,930 more rows, and 3 more
## #   variables: x <dbl>, y <dbl>, z <dbl>
```

Tibbles

The `diamonds` data is an example of a **tibble** object, which is a "trimmed down" version of a data frame. Tibbles are one of the central data structures used in the tidyverse.

Tibbles are data frames with the added class **`tbl_df`** that makes the way R prints the object more readable (e.g., typing `diamonds` did not print the entire data frame with 53940 rows).

The **`tibble()`** function can create tibbles, the same way that `data.frame()` does.

The **`as_tibble()`** function coerces lists and matrices into tibbles.

Side Note: The `tbl_df` class and its basic functions are encapsulated in the `tibble` package, but all tidyverse packages will load `tibble` automatically.

Creating Tibbles

```
tb <- tibble(
  `:)` = "smile",
  `:(` = "sad",
  `2018_$` = 200L,
  p = 0.6
)
tb
```

```
## # A tibble: 1 x 4
##   `:)`  `:(`  `2018_$`     p
##   <chr> <chr>    <int> <dbl>
## 1 smile sad        200   0.6
```

Creating Tibbles

```
tb <- tribble(
  ~`:(`, ~Year, ~Saving,
#Sad?/12 months/an account
"Yes", 2017, 2,
"No", 2018, 2000,
)
tb
```

```
## # A tibble: 2 x 3
##   `:(`   Year Saving
##   <chr> <dbl>  <dbl>
## 1 Yes    2017      2
## 2 No     2018   2000
```

Tibbles Versus Data Frames

Since tibbles are data frames, the basic syntax and functions for data frames work for tibbles, so we will mostly treat them the same way.

Besides the cleaner printing, the main differences between tibbles and data frames are:

- Column data is not coerced. In particular, a character vector is not coerced into a factor.
- Subsetting a column from a tibble using the single bracket `[,j]` always returns a tibble rather than extracting the vector inside (i.e., `[,j,drop=FALSE]` is the default behavior).
- The `$` operator does not allow partial name matching the way it does for data frames. (e.g., `diamonds$cu` throws an error but `as.data.frame(diamonds)$cu` does not.)

Printing

Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on screen.

```
diamonds %>% print(n = 5, width = Inf)
```

```
## # A tibble: 53,940 x 10
##   carat cut     color clarity depth table price
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int>
## 1  0.23 Ideal   E     SI2      61.5    55   326
## 2  0.21 Premium E     SI1      59.8    61   326
## 3  0.23 Good    E     VS1      56.9    65   327
## 4  0.29 Premium I     VS2      62.4    58   334
## 5  0.31 Good    J     SI2      63.3    58   335
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1  3.95  3.98  2.43
## 2  3.89  3.84  2.31
## 3  4.05  4.07  2.31
```

Subsetting

```
tb$Year %>%
  identical(tb[["Year"]])
```

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

```
tb[[2]] %>%
  identical(tb[["Year"]])
```

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

```
class(diamonds[,1])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
class(iris[,1])
```

```
## [1] "numeric"
```

Section 3

Data Import with readr

Importing Rectangular Data

The `readr` package contains functions to import plain-text rectangular data into R.

One of the most common file types for plain-text rectangular data is the CSV (comma separated values) file.

The most common function in base R to import data from CSV files is **`read.csv()`** (which is a wrapper function for **`read.table()`**).

The `readr` version is the **`read_csv()`** function.

The Main Functions of `readr`

There are several versions of the `read_*()` functions for different types of delimiters/separators:

- **`read_csv()`** reads comma delimited files.
- **`read_csv2()`** reads semicolon delimited files.
- **`read_tsv()`** reads tab delimited files.
- **`read_delim()`** reads files with any delimiter.
- **`read_fwf()`** reads fixed width files.
- **`read_table()`** reads files with white space separators.

The first argument of all of these functions is the `file` argument, which is the name and location of the import file (in quotations).

Since all of these functions have similar syntax, we will focus on the `read_csv()` function.

read_csv() Example

```
births <- read_csv("births.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   weight = col_double(),
##   Apgar1 = col_double(),
##   Fage = col_double(),
##   Mage = col_double(),
##   Feduc = col_double(),
##   Meduc = col_double(),
##   TotPreg = col_double(),
##   Visits = col_double(),
##   Gained = col_double()
## )

## See spec(...) for full column specifications.
```

read_csv() Example

```r
head(births)
```

```
## # A tibble: 6 x 21
##   Gender Premie weight Apgar1  Fage  Mage Feduc
##   <chr>  <chr>   <dbl>  <dbl> <dbl> <dbl> <dbl>
## 1 Male   No        124      8    31    25    13
## 2 Female No        177      8    36    26     9
## 3 Male   No        107      3    30    16    12
## 4 Female No        144      6    33    37    12
## 5 Male   No        117      9    36    33    10
## 6 Female No         98      4    31    29    14
## # ... with 14 more variables: Meduc <dbl>,
## #   TotPreg <dbl>, Visits <dbl>, Marital <chr>,
## #   Racemom <chr>, Racedad <chr>, Hispmom <chr>,
## #   Hispdad <chr>, Gained <dbl>, Habit <chr>,
## #   MomPriorCond <chr>, BirthDef <chr>,
## #   DelivComp <chr>, BirthComp <chr>
```

Tibble Output

A few things to note:

- The `read_csv()` function prints the column specification with the name and type of each column. This can be helpful to make sure the file is read correctly.
- The output of `read_csv()` is always a tibble object.
- The first line is read as the column names by default.
- Character columns are not coerced into factors.

Inline Data

The `read_csv()` function also supports inputting data inline.

```
read_csv("a,b,c
         1,2,3
         4,5,6")
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

```
read_csv("a, b, c \n 1, 2, 3\n 4, 5, 6") # same thing
```

Note: The **\n** inside a character/string represents a line break.

Optional Arguments in `read_csv()`

There are several optional arguments in the `read_csv()` function that can be important to know for certain scenarios:

- Use **`skip = n`** to skip the first `n` lines (e.g., if there is metadata at the top of the file).
- Use **`comment = "#"`** to drop all lines starting with the # character.
- Use **`col_names = FALSE`** to read files without column names. The columns will be labeled sequentially from `X1` to `Xn` (for `n` columns).
- Alternatively, input a character vector in **`col_names`** to specify column names.
- Use the **`na`** argument to specify the character(s) that represent missing values in the file.

The fread() Function

For most purposes, the `read_csv()` function is almost universally preferred over the base R `read.csv()` function.

In addition to the benefit of outputting tibbles instead of data frames, the `readr` functions are typically much faster (around 10x[1]) than the base R versions.

However, for extremely large datasets (e.g., gigabytes of data with millions or even billions of rows), the **`data.table::fread()`** function is much faster than even the `readr` functions.

[1] According to Grolemund and Wickham's "R for Data Science".

The `data.table` Package

The **`data.table`** package is outside the tidyverse, and it has its own syntax that is outside the scope of this course. But it is important to know it exists as an alternative for dealing with large data.

For more information on `data.table`:
https://github.com/Rdatatable/data.table/wiki

Section 4

Tidy Data with tidyr

Tidy Data

Almost all the functions and packages in the tidyverse assume that your data is organized in a consistent way that is meant to make the data easier to manipulate and visualize.

Data organized in this way is called **tidy data**.

Hadley Wickham's paper on the underlying theory and motiviation for tidy data: http://www.jstatsoft.org/v59/i10/paper

Data Example 1

The following datasets contain the number of TB cases documented by the World Health Organization in Afghanistan, Brazil, and China between 1999 and 2000.

The objects used here are found in the tidyr package.

```
table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Data Example 2

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

Data Example 3

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

Data Example 4

```
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```
table4b
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
## * <chr>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

Data Example

Each dataset shows the same values for the same four variables (country, year, population, and cases), but each dataset organizes the values in a different way.

Even though the underlying data is the same for each dataset, some representations of the data are easier to work with than others.

The tidyverse works well with the tidy representation of data. We need to be able to recognize when data is tidy and how to reorganize data into tidy format.

Tidy Data Rules

A dataset is called **tidy** if the following three rules are satisfied:

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.

variables observations values

Any dataset that is not tidy is sometimes called **messy**.

Question: Which of the preceding tibbles is tidy?

The Main Functions of `tidyr`

The first step in tidying data is to figure out what the **variables** and **observations** are.

The main functions in the `tidyr` package help make data tidy.

Once the variables and observations are specified, there are several functions that address common issues with messy data:

- **`gather()`** is used when one variable is spread across multiple columns.
- **`spread()`** is used when one observation is scattered across multiple rows.
- **`separate()`** is used when cells contain multiple values (from different variables).
- **`unite()`** is used when a single variable is spread across multiple columns.

Common Syntax

All of the `tidyr` functions have the same basic syntax:

1. The first argument is a data frame (or tibble).
2. Subsequent arguments describe what to do with the data frame (variable names can be used without quotations).
3. The output is a data frame (same class as the input).

Gathering

One example of when a variable is spread across multiple columns is where column names are not names of variables but values of a variable.

For example, consider `table4a`:

```
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

The 1999 and 2000 column names represent values of the `year` variable. We need to **gather** the two year columns into a new pair of variables.

The gather() Function

The **`gather()`** function gathers multiple columns and collapses them into **key-value pairs**:

- The **`key`** is the name of the variable whose values form the column names.
- The **`value`** is the name of the variable whose values are spread over the cells.

The gather() Function

For example:

```
table4a_demo <- table4a %>%
  gather(`1999`, `2000`, key="year", value="cases")
table4a_demo %>%
  spread(key = "year", value = "cases")
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
##   <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

Note: To use non-standard (or **non-syntactic**) column names, we need to include backticks.

Spreading

Spreading is the opposite of gathering.

For example, consider `table2`:

```
head(table2, 4)
```

```
## # A tibble: 4 x 4
##   country      year type           count
##   <chr>       <int> <chr>          <int>
## 1 Afghanistan  1999 cases            745
## 2 Afghanistan  1999 population  19987071
## 3 Afghanistan  2000 cases           2666
## 4 Afghanistan  2000 population  20595360
```

When an observation is scattered across multiple rows, we want to **spread** the observation from narrow/stacked rows into one wider row.

The spread() Function

The **spread()** function spreads a key-value pair across multiple columns.

- The **key** is the column that contains variable names.
- The **value** is the column that contains the values from multiple variables.

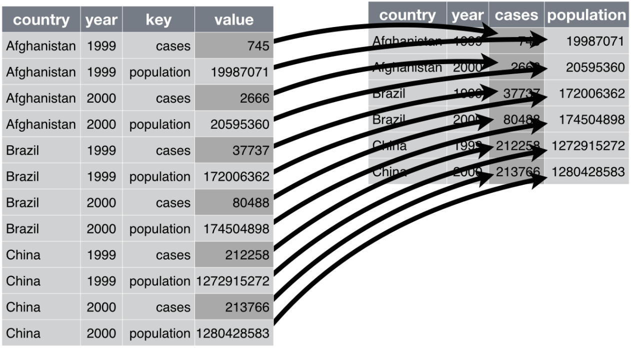

table2

The spread() Function

For example:

```r
table2 %>%
  spread(key = "type", value = "count")
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Separating

Another issue that can arise with non-tidy data is when cells contain multiple values.

For example, consider `table3`:

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

We want to **separate** the values from the `rate` variable into two variables, `cases` and `population`.

The separate() Function

The **separate()** function pulls apart one column into multiple columns, by splitting wherever a separator character appears.

```
table3 %>%
  separate(rate,into=c("cases", "population"))
```

```
## # A tibble: 6 x 4
##   country      year cases  population
##   <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

The **into** argument specifies the names of the columns to split the input column into. The separator can be specified using the optional **sep** argument (by default it will separate by any non-alphanumeric character).

Uniting

The opposite of separate() is **unite()**.

For example, consider `table5`:

```
table5
```

```
## # A tibble: 6 x 4
##   country     century year  rate
## * <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

The year variable is split into `century` and `year` columns.

The unite() Function

The **unite()** function combines multiple columns into a single column.

The **col** argument is the name of the new variable we want to create. The columns that will be combined are then inputted as separate arguments.

```
table5 %>%
  unite(new, century, year, sep = "")
```

```
## # A tibble: 6 x 3
##   country     new   rate
##   <chr>       <chr> <chr>
## 1 Afghanistan 1999  745/19987071
## 2 Afghanistan 2000  2666/20595360
## 3 Brazil      1999  37737/172006362
## 4 Brazil      2000  80488/174504898
## 5 China       1999  212258/1272915272
## 6 China       2000  213766/1280428583
```

The optional **sep** argument specifies the separator to insert between the combined values. The default is an underscore: `sep="_"`.