An Introduction to the Tydyverse

Neftali Lemus

1 Introduction

Tidyverse is a system of R packages to improve data management, exploration and visualization in R.

1.1 Core Tidyverse Packages

- tibble: A more efficient data frame.
- readr: Read rectangular data.
- tidyr: Make data "tidy".
- dplr: Data manipulation.
- purrr: Enhace R's functional programming.
- stringr: String (character) manupulation.
- forcats: Factors (categorical variables)
- ggplot: Graphics system

```
library(tidyverse)
```

1.2 Pipe

The %>% operator places left hand objects into the first argument of the right hand side function.

Shortcut: Ctrl + Shift + M = %>%

f(x) can also be written as x %>% f()

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

x %>% f(y) %>% g(z) can also be written as g(f(x,y),z).

1.3 Pipe Examples

```
pi %>% cos
```

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

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

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

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

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

1.4 Argument Placeholder .

f(x) can also be written as f(x,.)

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

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

2 Tibbles

2.1 Tibble Example

Using the *diamonds* data in ggplot2.

```
data("diamonds")
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

2.2 Tibbles

- *diamonds* data is a **tibble** object.
- Tibbles are data frames with the added class *tbl_df*. Print is more readable.
- tibble() creates tibbles while data.frame() creates data frames.
- as_tibble() coerces lists and matrices into tibbles.

2.3 Creating Tibbles

```
# 1st Way
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
```

```
# 2nd Way
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
```

2.4 Tibbles Versus Data Frames

- Basic syntax and functions for data frames work for tibbles.

Main Differences:

- 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.)

2.5 Printing

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

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

```
## # A tibble: 53,940 x 10
##   carat cut     color clarity depth table price     x     y     z
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal   E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium E     SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good    E     VS1      56.9    65   327  4.05  4.07  2.31
## 4  0.29 Premium I     VS2      62.4    58   334  4.2   4.23  2.63
## 5  0.31 Good    J     SI2      63.3    58   335  4.34  4.35  2.75
## # ... with 53,935 more rows
```

2.6 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"
```

3 Data Import with readr

- The *readr* package contains functions to import plain-text rectangular data into R (csv).
- read.csv() is base R
- The *readr* version is the read_csv() function.

3.1 Main Functions of readr

- 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.

3.2 read_csv() Example

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

```
## # A tibble: 6 x 21
##   Gender Premie weight Apgar1  Fage  Mage Feduc Meduc TotPreg Visits Marital
##   <chr>  <chr>   <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl>  <dbl> <chr>
## 1 Male   No        124      8    31    25    13    14       1     13 Married
## 2 Female No        177      8    36    26     9    12       2     11 Unmarried
## 3 Male   No        107      3    30    16    12     8       2     10 Unmarried
## 4 Female No        144      6    33    37    12    14       2     12 Unmarried
## 5 Male   No        117      9    36    33    10    16       2     19 Married
## 6 Female No         98      4    31    29    14    16       3     20 Married
## # ... with 10 more variables: Racemom <chr>, Racedad <chr>, Hispmom <chr>,
## #   Hispdad <chr>, Gained <dbl>, Habit <chr>, MomPriorCond <chr>,
## #   BirthDef <chr>, DelivComp <chr>, BirthComp <chr>
```

3.3 Tibble Output

- 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.

3.4 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
```

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

3.5 Optional Arguments in Read_csv()

- 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.

3.6 The fread() Function

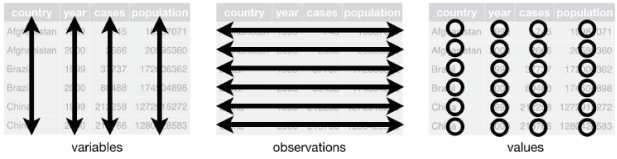
- The **readr functions are typically much faster** (around $10x$[1]) than the base R versions.
- For extremely large datasets (e.g., gigabytes of data with millions or even billions of rows), the *data.table::fread()* function is much faster than even the readr functions.

4 Tydy Data with tidyr

Data organized in a consistent way that is meant to make the data easier to manipulate and visualize is called **tidy data**.

4.1 Tidy Data Rules

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



4.2 The Main Functions of tydyr

- 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.

4.3 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.

```
table4a
```

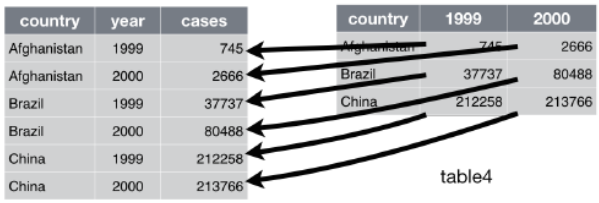
```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

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.

4.4 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.



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.

4.5 Spreading

Spreading is the opposite of gathering.

```
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.

4.6 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.

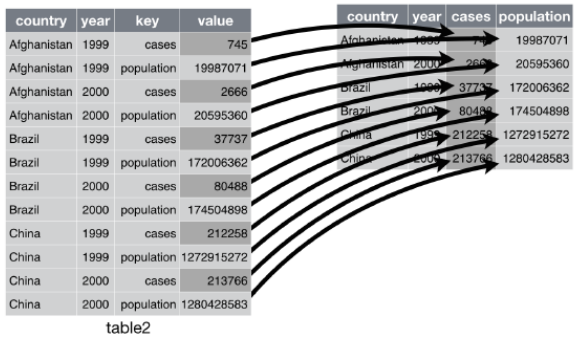


table2

For example:

```
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
```

4.7 Separating

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

```
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.

4.8 The separating() Function

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

```
## # A tibble: 6 x 4
##   country      year cases  pupulation
##   <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).

4.9 Uniting

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

For example:

```
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.

4.10 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="_".