Introduction to the Tidyverse

Max Turgeon

DATA 2010–Tools and Techniques in Data Science

Lecture Objectives

- Choose the right `tidyverse` function for data transformation
    - Use the pipe operator to chain function calls

Motivation

- Last lecture, we used `R` to compute summary statistics.
    - We only used *base* `R`
- But we quickly saw some limitations of this approach.
    - Summarizing by group was tedious
- Today we will introduce the `tidyverse`, which is a suite of packages that make it easier to work with data frames

Transforming your data

- Sometimes, you want to look at a subset of the data. Or perhaps you want to compute the mean of another variable, not defined in your dataset.
    - In other words, we need to transform the data first!
- All `tidyverse` functions take a `data.frame` as the first argument.
- A `data.frame` is a collection of vectors, all of the same length, but could be of different types.
    - This is the main way of organizing data in `R`.

`tidyverse` packages

- The main `tidyverse` packages are:
  - `dplyr`: Main data transformation functions
  - `tidyr`: To turn your data into the tidy format (more on this later)
  - `readr`: Import data into `R`
  - `ggplot2`: Data visualization
- All these packages can loaded by calling `library(tidyverse)`.
- These packages are maintained by RStudio.
- In Python, you would use Pandas for data manipulation/transformation.

Main tidyverse functions i

- `mutate`: Create a new variable as a function of the other variables

```
# Switch to litres per 100km
mutate(mtcars, litres_per_100km = 235.215/mpg)
```

- `filter`: Keep only rows for which some condition is `TRUE`

```
# Only keep rows where cyl is equal to 6 or 8
filter(mtcars, cyl %in% c(6, 8))
```

Example i

- Let's say we want to compute a 95% confidence interval for litres per 100km.

```r
library(tidyverse)

data1 <- mutate(mtcars, litres_per_100km = 235.215/mpg)
data2 <- summarise(data1,
                   avg_lit = mean(litres_per_100km),
                   sd_lit = sd(litres_per_100km))
data2
```

Example ii

```
##    avg_lit   sd_lit
## 1 12.75506 3.863251
```

```
n <- nrow(mtcars)
data3 <- mutate(data2,
                low_bd = avg_lit - 1.96*sd_lit/sqrt(n),
                up_bd = avg_lit + 1.96*sd_lit/sqrt(n))
data3
```

```
##    avg_lit   sd_lit   low_bd    up_bd
## 1 12.75506 3.863251 11.41651 14.09361
```

Exercise

Use the `gapminder` dataset from the package `dslabs` to compute the average life expectancy across all countries for the year 2016. Compute a 95% confidence interval for this average.

Solution i

```r
library(dslabs)

data1 <- filter(gapminder,
                year == 2016)
data2 <- summarise(data1,
                   avg_le = mean(life_expectancy),
                   sd_le = sd(life_expectancy))
```

Solution ii

```
n <- nrow(data1)
data3 <- mutate(data2,
                low_bd = avg_le - 1.96*sd_le/sqrt(n),
                up_bd = avg_le + 1.96*sd_le/sqrt(n))
data3
```

```
##     avg_le    sd_le   low_bd   up_bd
## 1 72.46757 7.584449 71.37463 73.5605
```

Pipe operator

- One of the important features of the `tidyverse` is the pipe operator `%>%`
- It takes the output of a function (or of an expression) and uses it as input for the next function (or expression)

```r
library(tidyverse)

count(mtcars, cyl)
```

```
##   cyl  n
## 1   4 11
## 2   6  7
## 3   8 14
```

```r
# Or with the pipe
# mtcars becomes the first argument of count
mtcars %>% count(cyl)
```

Pipe operator

- In more complex examples, with multiple function calls, the pipe operator improves readability.

```
# Without pipe operator
fit_model(prepare_data(dataset))
# With pipe operator
dataset %>%
  prepare_data %>%
  fit_model
```

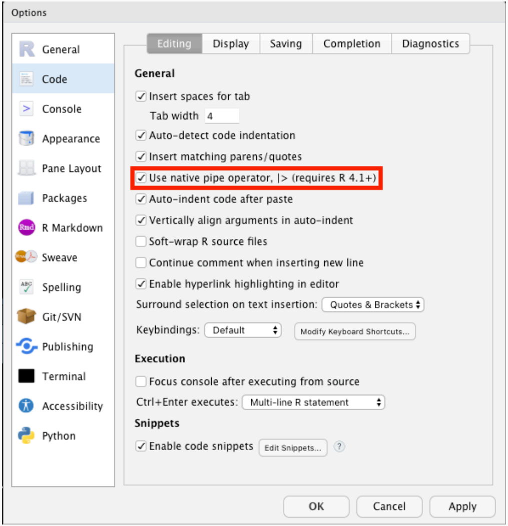
New in R 4.1.1

- If you just installed or updated R, there is a new native pipe operator!
  - It's `|>` instead of `%>%`.
- They are essentially equivalent, but be careful:
  - `vect %>% mean` is valid, but `vect |> mean` will throw an error.
- You **always** need the parentheses with the native pipe.
  - `vect |> mean()`

Pipe in RStudio i

- There is a handy shortcut for the pipe operator if you are using Rstudio:
  - `Cmd + Shift + m` on Mac
  - `Ctrl + Shift + m` on Windows/Linux
- With the newest version of RStudio, you can choose whether to print the `tidyverse` or the native pipe.

Pipe in RStudio ii



General

☑ Insert spaces for tab

Tab width 4

☑ Auto-detect code indentation

☑ Insert matching parens/quotes

☑ Use native pipe operator, |> (requires R 4.1+)

☑ Auto-indent code after paste

☑ Vertically align arguments in auto-indent

☐ Soft-wrap R source files

☐ Continue comment when inserting new line

☑ Enable hyperlink highlighting in editor

Surround selection on text insertion: Quotes & Brackets

Keybindings: Default

Execution

☐ Focus console after executing from source

Ctrl+Enter executes: Multi-line R statement

Snippets

☑ Enable code snippets

Examples i

```
# Let's convert our previous example to use the pipe
mtcars %>%
  mutate(litres_per_100km = 235.215/mpg) %>%
  summarise(avg_lit = mean(litres_per_100km),
            sd_lit = sd(litres_per_100km)) %>%
  mutate(low_bd = avg_lit - 1.96*sd_lit/sqrt(n),
         up_bd = avg_lit + 1.96*sd_lit/sqrt(n))
```

```
##    avg_lit   sd_lit   low_bd    up_bd
## 1 12.75506 3.863251 12.19836 13.31176
```

Examples ii

- We didn’t need intermediate datasets `data1`, `data2` and `data3`.
- It’s easier to read.

Summaries by group

- We can combine `summarise` and `group_by` to create summaries for each group individually.

```
# Average mpg for each value of cyl
mtcars %>%
  group_by(cyl) %>%
  summarise(avg_mpg = mean(mpg))
```

```
## # A tibble: 3 x 2
##     cyl avg_mpg
##   <dbl>   <dbl>
## 1     4    26.7
## 2     6    19.7
## 3     8    15.1
```

Examples i

```r
# Average mpg for each value of cyl + 95% CI
mtcars %>%
  group_by(cyl) %>%
  summarise(avg_mpg = mean(mpg),
            sd_mpg = sd(mpg),
            n = n()) %>%
  mutate(low_bd = avg_mpg - 1.96*sd_mpg/sqrt(n),
         up_bd = avg_mpg + 1.96*sd_mpg/sqrt(n))
```

Examples ii

```
## # A tibble: 3 x 6
##     cyl avg_mpg sd_mpg     n low_bd up_bd
##   <dbl>   <dbl>  <dbl> <int>  <dbl> <dbl>
## 1     4    26.7   4.51    11   24.0  29.3
## 2     6    19.7   1.45     7   18.7  20.8
## 3     8    15.1   2.56    14   13.8  16.4
```

- **Very important**: The number of observations in each group is different!
- This is why we computed the number of observations in each group using the function `n()`.

Examples iii

- When we compute the confidence interval, the variable `n` refers to the column `n` in the dataset, i.e. what we computed using summarise.
- Here, the "word" `n` refers to three different things:
    - A function, `n()`, which counts the number of observations.
    - A column in the dataset that we created using the function `n()`.
    - The number of rows of `mtcars` that we computed earlier.
- R keeps track of all of these (using something called "scoping rules"), but for a human this can be confusing... It's best to avoid it if we can.

Examples iv

```
# Average mpg for each value of cyl + 95% CI
mtcars %>%
  group_by(cyl) %>%
  summarise(avg_mpg = mean(mpg),
            sd_mpg = sd(mpg),
            nobs = n()) %>%
  mutate(low_bd = avg_mpg - 1.96*sd_mpg/sqrt(nobs),
         up_bd = avg_mpg + 1.96*sd_mpg/sqrt(nobs))
```

Exercise

Compute the average life expectancy by continent for 2016 using the `gapminder` dataset. Compute 95% confidence intervals.

Solution i

```r
gapminder %>%
  filter(year == 2016) %>%
  group_by(continent) %>%
  summarise(avg_le = mean(life_expectancy),
            sd_le = sd(life_expectancy),
            nobs = n()) %>%
  mutate(low_bd = avg_le - 1.96*sd_le/sqrt(nobs),
         up_bd = avg_le + 1.96*sd_le/sqrt(nobs))
```

Solution ii

```
## # A tibble: 5 x 6
##   continent avg_le sd_le  nobs low_bd up_bd
##   <fct>      <dbl> <dbl> <int>  <dbl> <dbl>
## 1 Africa      63.8  6.17    51   62.1  65.5
## 2 Americas    75.2  3.54    36   74.1  76.4
## 3 Asia        74.8  5.13    47   73.3  76.3
## 4 Europe      78.9  3.37    39   77.9  80.0
## 5 Oceania     71.0  7.23    12   66.9  75.0
```