COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

An introduction to the tidyverse

Contents

Why using tidyverse?

- All (or maybe most of) `tidyverse` functions can be done using base R, so why using it?
- Because it offers better consistency
  - Functions in base R have **inconsistencies** in both the interfaces to the functions, as well as their **outputs**.
  - This can both slow down learning and inefficiencies in the code.
- However base R still has its advantages, so stay informed about all the options!

comillas.edu

Why using tidyverse?

- Tidyverse is composed of several libraries each with its purpose:
  - Data wrangling: `dplyr`, `tidyr`, `readr`
  - Iteration: `purrr`
  - Visualization: `ggplot2`

purr
The map family

purr offers a family of map functions as an alternative base R's apply functions:

- map(): apply a function across elements of a list or vector → list in, list out
- map_dbl(), map_int(), map_lgl(), map_chr(): same, but return a vector of a particular data type → list in, vector out (double, integer, logical or character vector)
- map_dfr(), map_dfc(): same, but combines the output list into a data.frame. They differ in whether that data.frame is formed by **r**ow-binding or **c**olumn-binding → list in, data.frame out

All this functions are used following map(x, f) where x is a list or vector and f is the function applied to all members of x.

dplyr
What is dplyr

`map_dfr()` and `map_dfc()` functions depend on another package called `dplyr`. `dplyr` is another tidyverse package very useful for data.frame computations. It provides the tidyverse alternative to the some base R functions:

- `mutate()` adds new variables that are functions of existing variables
- `mutate_at()` apply functions to given columns
- `select()` picks variables based on their names.
- `filter()` picks cases based on their values.
- `summarize()` reduces multiple values down to a single summary.
- `count()` count number of rows in each group.
- `arrange()` changes the ordering of the rows.
- `group_by()` allows to perform any operation "by group"
- %>% pipe operator, elegant glue functionality together

`dplyr` functions are analogous to SQL counterparts, so learn `dplyr` and get SQL for free!

dplyr
All behold the glorious pipe

- Tidyverse functions are at their best when composed together using the pipe operator %>%. Use `ctrl/cmd + shift + m` as **shortcut** in RStudio.
- This operator actually comes from the `magrittr` package (automatically loaded with `dplyr`).
- A literal translation of piping:

Take one return value and automatically feed it in as an input to the next function to form a flow of results

- For example: `x %>% f %>% g %>% h` is the same as `h(g(f(x)))`

Hint: in your mind, when you see %>% read it as "and then"

dplyr
All behold the glorious pipe

- For multi-arguments functions: `x %>% f(y)` is the same as `f(x,y)`
  - Note that it x is passed in the first argument. But how can it be passed to other positions? Using **dot notation**.
  - `x %>% f(y)` can be written as `x %>% f(., y)`
  - But to pass x in the *second* argument: `x %>% f(y, .)`
- A more complicated example:

```
x = "Prof Jaime really loves piping"
x %>%
  strsplit(split = " ") %>%
  .[[1]] %>% # indexing, could also use `[[`(1)
  nchar %>%
  max
## [1] 6
```

dplyr
Ordering

- `arrange()` function is used to order rows by values of a column

```
mtcars %>%
  arrange(mpg) %>%
  head(4)
```

- It can also be ordered by descending order:

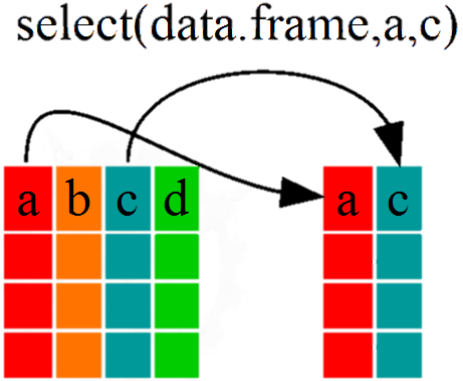
```
mtcars %>%
  arrange(desc(mpg)) %>%
  head(4)
```

- Or ordered by multiple columns:

```
mtcars %>%
  arrange(desc(gear), desc(hp) %>%
  head(4)
```

dplyr
Manipulate variables

- `select()` function is used to create a data.frame equal to subset of columns from a data.frame



- Using pipes:

```
year_country_gdp <- gapminder %>% select(year, country, gdpPercap)
```

dplyr
select() helpers

```
mtcars %>% select(starts_with("d")) %>% head(2)
    ##               disp drat
    ## Mazda RX4      160  3.9
    ## Mazda RX4 Wag  160  3.9
mtcars %>% select(ends_with('t')) %>% head(2)
    ##               drat    wt
    ## Mazda RX4      3.9 2.620
    ## Mazda RX4 Wag  3.9 2.875
mtcars %>% select(ends_with('yl')) %>% head(2)
    ##               cyl
    ## Mazda RX4       6
    ## Mazda RX4 Wag   6
mtcars %>% select(contains('ar')) %>% head(2)
    ##               gear carb
    ## Mazda RX4        4    4
    ## Mazda RX4 Wag    4    4
```

You can find more helpers functions here

dplyr
Manipulate cases

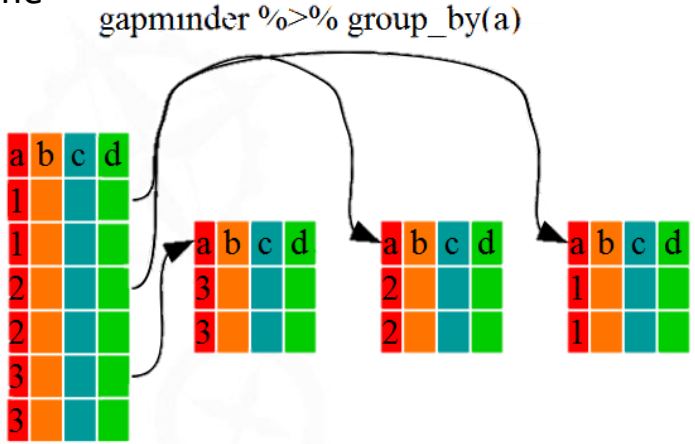
- `filter()` function is used to create a data.frame equal to subset of rows that match a condition from a data.frame
- Using pipes:

```
year_country_gdp <- gapminder %>%
            filter(continent=="Europe") %>%
            select(year, country, gdpPercap)
```

- Note that the order of operations is very important in this case. If `select()` is used first, the `filter()` function would not have been able to find the variable "continent" because it would have been removed in the previous step.

dplyr
Grouping data

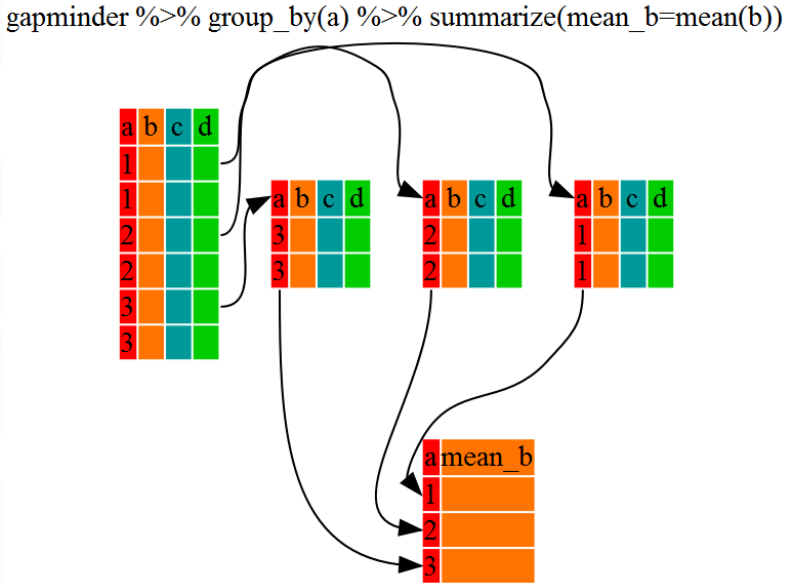
- `group_by()` allows to define groups of rows based on columns or conditions.
- This does not actually change anything about the way the data.frame looks, but it play a big role in how `dplyr` functions act on the data.frame



- A data.frame can be grouped by several variables at the same time as `dataframe %>% group_by(a,b)`

dplyr
Applying operations on (groups of) data

- `summarize()` apply user-defined operations to the rows of a data.frame.
- If the data.frame is grouped using the `group_by()` function, the operations are isolatedly applyed on the groups of data



dplyr
Counting

- `count(data, …)` count the number of rows in each group defined by the variables in …

```
gapminder %>%
    filter(year == 2002) %>%
    count(continent, sort = TRUE)
```

- `n()` count the number of rows in each group to be used inside the pipe operations.

```
gapminder %>%
    group_by(continent) %>%
    summarize(
      mean_le = mean(lifeExp),
      min_le = min(lifeExp),
      max_le = max(lifeExp),
      se_le = sd(lifeExp)/sqrt(n()))
```

dplyr
Creating new variables

- New variables can be created before (or even after) summarising the information using `mutate()`.
- In the example, a new variable is created and then used to summarize the data.

```
gdp_pop_bycontinents_byyear <- gapminder %>%
    mutate(gdp_billion=gdpPercap*pop/10^9) %>%
    group_by(continent,year) %>%
    summarize(mean_gdpPercap=mean(gdpPercap),
              sd_gdpPercap=sd(gdpPercap),
              mean_pop=mean(pop),
              sd_pop=sd(pop),
              mean_gdp_billion=mean(gdp_billion),
              sd_gdp_billion=sd(gdp_billion))
```

dplyr
Conditionally creating new variables

- New variables can be created following a logical condition using a combination of `mutate()` and `ifelse()`:

```
# Creating new variable and filtering rows based on condition
gdp_pop_bycontinents_byyear_above25 <- gapminder %>%
    mutate(gdp_billion = ifelse(lifeExp > 25, gdpPercap *
           pop / 10^9, NA)) %>%
    group_by(continent, year) %>%
    summarize(mean_gdpPercap = mean(gdpPercap),
              sd_gdpPercap = sd(gdpPercap),
              mean_pop = mean(pop),
              sd_pop = sd(pop),
              mean_gdp_billion = mean(gdp_billion),
              sd_gdp_billion = sd(gdp_billion))
```

dplyr
Apply function to column

- `mutate_at()` apply a function to one or several columns:

```
mtcars = mtcars %>%
    mutate_at(c("hp_wt", "mpg_wt"), log)

# Base R
mtcars$hp_wt = log(mtcars$hp_wt)
mtcars$mpg_wt = log(mtcars$mpg_wt)
```

dplyr
dplyr vs SQL

- Once you learn dplyr you should find SQL very natural, and vice versa!
- For example, `select` is `SELECT`, `filter` is `WHERE`, `arrange` is `ORDER BY` etc.
- This will make it much easier for tasks that require using both R and SQL to munge data and build statistical models
- One major link is through powerful verbs like `group_by()` and `summarize()`, which are used to aggregate data
- Another major link to SQL is through merging/joining data frames, via `left_join()` and `inner_join()` verbs

dplyr Cheatsheet

- Cheatsheet of the dplyr package can be found here

`tidyr`
Main functions

Two of the most important `tidyr` verbs are:

- `pivot_longer()`: make "wide" data longer
- `pivot_wider()`: make "long" data wider

There are many others like `spread()`, `gather()`, `nest()`, `unnest()`, etc… which you could find here

tidyr
pivot_longer()

```
EDAWR::cases
##   country  2011  2012  2013
## 1      FR  7000  6900  7000
## 2      DE  5800  6000  6200
## 3      US 15000 14000 13000
```

```
EDAWR::cases %>% pivot_longer(names_to = "year",
                              values_to = "n",
                              cols = 2:4)
```

```
## # A tibble: 9 × 3
##   country year      n
##   <chr>   <chr> <dbl>
## 1 FR      2011   7000
## 2 FR      2012   6900
## 3 FR      2013   7000
## 4 DE      2011   5800
## 5 DE      2012   6000
## 6 DE      2013   6200
## 7 US      2011  15000
## 8 US      2012  14000
## 9 US      2013  13000
```

tidyr
pivot_wider()

```
EDAWR::pollution
##       city  size amount
## 1 New York large     23
## 2 New York small     14
## 3   London large     22
## 4   London small     16
## 5  Beijing large    121
## 6  Beijing small     56

EDAWR::pollution %>% pivot_wider(names_from = "size",
                values_from = "amount")
## # A tibble: 3 x 3
##   city     large small
##   <chr>    <dbl> <dbl>
## 1 New York    23    14
## 2 London      22    16
## 3 Beijing    121    56
```

Bibliography

- H. Wickham and G. Grolemund (2021). *R for Data Science*.
- L. Wilkinson (2003). *The Grammar of Graphics* (Statistics and Computing). 2nd Ed. Springer.
- R. Tibshirani (2021). *Statistical Computing Course: Fall 2001*. URL: http://www.stat.cmu.edu/~ryantibs/statcomp/
- Tidyverse (2021). URL: https://www.tidyverse.org/