Introduction to R

Tidyverse-I

Sumit Mishra

Krea University | WSDS002

Contents

Tibbles

`tibble` is the modern version of `R`'s `data.frame`.

Example `starwars` dataset that is a built-in tibble.

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

name	height	mass	sex	gender	homeworld
Luke Skywalker	172	77	male	masculine	Tatooine
C-3PO	167	75	none	masculine	Tatooine
R2-D2	96	32	none	masculine	Naboo
Darth Vader	202	136	male	masculine	Tatooine
Leia Organa	150	49	female	feminine	Alderaan
Owen Lars	178	120	male	masculine	Tatooine

Creation

The `tibble` function creates a `data.frame` like-object. You'll generally define tibble by passing the column names and values for the columns.

```
album <- c("Please Please Me", "Rubber Soul", "Magical Mystery Tour")
year <-  c(1963, 1965, 1967)
num.tracks <- c(14,14,11)
beatles.catalog <- tibble(album, year, num.tracks)
```

You can also create tibbles from other objects (*e.g.*, matrices) using the function `as_tibble()`[1]

Importing a dataset

- `read_csv`: reads csv file into R as a tibble.

```
g3 <- read_csv("../data/gdp-growth.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   `Country Name` = col_character(),
##   `Country Code` = col_character(),
##   `Series Name` = col_character()
## )
```

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

```
head(g3)
```

```
## # A tibble: 6 x 23
##   `Country Name` `Country Code` `Series Name` YR1999 YR2000 YR2001 YR2002 YR2003
##   <chr>          <chr>          <chr>          <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 Afghanistan    AFG            GDP growth (~  NA     NA     NA     NA     8.83
## 2 Albania        ALB            GDP growth (~  12.9    6.95   8.29   4.54  5.53
## 3 Algeria        DZA            GDP growth (~   3.20   3.82   3.01   5.61  7.20
## 4 American Samoa ASM            GDP growth (~  NA     NA     NA     NA     0.814
## 5 Andorra        AND            GDP growth (~   4.10   3.53   4.55   6.47 12.2
## 6 Angola         AGO            GDP growth (~   2.18   3.05   4.21  13.7   2.99
## # ... with 15 more variables: YR2004 <dbl>, YR2005 <dbl>, YR2006 <dbl>,
## #   YR2007 <dbl>, YR2008 <dbl>, YR2009 <dbl>, YR2010 <dbl>, YR2011 <dbl>,
## #   YR2012 <dbl>, YR2013 <dbl>, YR2014 <dbl>, YR2015 <dbl>, YR2016 <dbl>,
## #   YR2017 <dbl>, YR2018 <dbl>
```

- `read_xls`: reads an excel file. You will need to have the package `readxl` installed and loaded.

```
col <- read_xls("../data/COL.xls")
```

```
## # A tibble: 65 x 13
##     year population y_pop y_pop_us_100 y_wkr y_wkr_us_100 growth_100   i_y   g_y
##    <dbl>      <dbl> <dbl>        <dbl> <dbl>        <dbl> <chr>      <dbl> <dbl>
##  1  1950       11.9  3143         21.5  9820         27.1 NaN         25.8  6.89
##  2  1951       12.3  3092         20.0  9786         26.1 -1.6431     25.1  7.74
##  3  1952       12.7  3196         20.4 10247         26.7 3.3111000~  26.2  7.76
##  4  1953       13.1  3364         20.8 10926         27.6 5.1102999~  27.5  8.68
##  5  1954       13.5  3571         22.6 11751         29.3 5.9764999~  30.7  8.22
##  6  1955       14.0  3591         21.5 11963         28.5 0.5639999~  31.1  8.43
```

[1] Or just plain, old `tibble()`.

```
##  7  1956       14.4  3557         21.2 11978         28.6 -0.951500~  31.0  8.03
##  8  1957       14.9  3436         20.4 11695         27.4 -3.469800~  28.6  7.22
##  9  1958       15.4  3319         20.2 11423         26.6 -3.444100~  25.0  7.60
## 10  1959       15.9  3405         19.7 11846         26.3 2.5419999~  25.0  7.22
## # … with 55 more rows, and 4 more variables: ed_att <chr>, nx_y <dbl>,
## #   x_m_y <dbl>, ex_rate <dbl>
```

Year	Population	Y/Pop	Y/Pop(us=100)	Y/Wkr
1950	11.9333	3143	21.5011	9820
1951	12.3176	3092	20.0197	9786
1952	12.7148	3196	20.3629	10247
1953	13.1254	3364	20.8274	10926
1954	13.5499	3571	22.5808	11751

- R is pretty versatile in getting all kinds of datasets. For example, you can use `fromJSON()` to read json files from the internet.

```
cov19district <- jsonlite::fromJSON("https://api.covid19india.org/state_district_wise.json", flatten=T)
```

dplyr

Intro

It's a package. `dplyr` is not installed by default, so you'll need to install it[2].

`dplyr` is part of the `tidyverse`, and it follows a grammar-based approach to programming/data work.

- `data` compose the subjects of your stories
- `dplyr` provides the *verbs* (action words): `filter()`, `mutate()`, `select()`, `group_by()`, `summarize()`, `arrange()`

Manipulating variables: `mutate()`

`dplyr` streamlines adding/manipulating variables in your data frame.

Function: `mutate(.data, ...)`

- **Required argument:** `.data`, an existing data frame
- **Additional arguments:** Names and values of the new variables
- **Output:** An updated data frame

Example Take the data frame

```
df <- tibble(x = seq(2,22, length.out = 6),
             y = sample(1:20, 6))
```

`mutate()` allows us to create many new variables with one call.

[2] or just `p_load(dplyr)` after loading `pacman`

Code:

```
mutate(.data = df,
  xy = x * y,
  y2 = y^2,
  y_x = round(y/x),
  is_y_min = y == min(y)
)
```

Output:

```
## # A tibble: 6 x 6
##       x     y    xy    y2   y_x is_y_min
##   <dbl> <int> <dbl> <dbl> <dbl> <lgl>
## 1     2    20    40   400    10 FALSE
## 2     6     1     6     1     0 TRUE
## 3    10    10   100   100     1 FALSE
## 4    14     5    70    25     0 FALSE
## 5    18    14   252   196     1 FALSE
## 6    22    16   352   256     1 FALSE
```

Please note that `mutate()` returns the original *and* new columns.

Pipes

Before we go further, let's take a detour to learn about an important operator in tidyverse: pipe `%>%`. A *pipe* in programming allows you to take the output of one function and plug it into another function as an argument/input.

`R`'s pipe specifically plugs the returned object to the left of the pipe into the first argument of the function on the right fo the pipe, *e.g.*,

```
seq(2,22,length.out = 6) %>% mean() %>% round()
```

```
## [1] 12
```

Pipes help avoid lots of nested functions, and increase the readability of our code.

Example We will randomly pick six numbers between 5 and 30, compute their average, and round off the average. Remember the workflow.

$$\text{Numbers} \longrightarrow \text{Sample} \longrightarrow \text{Average} \longrightarrow \text{Round off}$$

```
# Save each intermediate step
numbers <- 5:30
our_sample <- sample(numbers,6)
ave.num <- round(mean(our_sample))
# Lots of nesting
ave.num <- round(mean(sample(5:30,6)))
print(ave.num)
```

```
## [1] 16
```

```
# Piping ⍰
ave.num <- 5:30 %>% sample(6) %>% mean() %>% round()
print(ave.num)
```

```
## [1] 13
```

By default, `R` pipes the output from the LHS of the pipe intothe first argument of the function on the RHS of the pipe.

E.g., `x %>% rep(3)` is equivalent to `rep(x, size = 3)`.

If you want to pipe output into a different argument, you use a period (`.`).

Example Suppose that you have a vector `x` of length 100, and you want to generate a sample `y` of size 10. You can achieve this using pipe in the following different ways.

```
x <- rnorm(100)
```

- Option 1

```
y <- x %>% sample(.,10,replace=F)
print(y)
```

```
##  [1]  0.9204039  0.6016666 -0.6007949  2.2084500 -0.1421788 -0.4025734
##  [7] -0.1024816  0.6042069  2.4706415  0.8028889
```

- Option 2

```
y <- 10 %>% sample(x,., replace = F)
print(y)
```

```
##  [1]  0.9204039 -0.1383846 -0.6007949 -0.9526688  0.4776354 -0.3674844
##  [7] -0.1065338  0.7058072 -1.1933317  1.0167047
```

- Option 3

```
y <- (replace = F) %>% sample(x,10,.)
print(y)
```

```
##  [1] -1.53744735  0.23736545  0.39136145  0.62136298 -0.28991560 -0.29567242
##  [7]  0.57790658 -0.75504410 -0.28112169 -0.08260369
```

%>% and dplyr

Each dplyr function begins with a .data argument so that you can easily pipe in data frames (recall: mutate(.data, …)).

The common workflow in dplyr will look something like

`new_df <- old_df %>% mutate(cool stuff here)`

which takes old_df, does some cool stuff with mutate(), and then saves the output of mutate() as new_df. Saving as a new (or replace the old) data frame helps you use the newly created columns.

Example

Without pipe:

```
new_df <-
  mutate(.data = df,
         xy = x * y,
         y2 = y^2,
         y_x = round(y/x),
         is_y_min = y == min(y)
)
```

Pipe:

```
# show output
new_df <- df %>%
  mutate(xy = x * y,
         y2 = y^2,
         y_x = round(y/x),
         is_y_min = y == min(y)
         )
```

select()

Just as filter() outputs row-based subsets of your tibble, select() grabs **column-based subsets**.

You can select columns using their **names** new_df %>% select(xy, x)

or you can select columns using **helper fuctions** new_df %>% select(starts_with("x"))

You can also choose to drop a column by prefixing the name of the column by hyphen (—).

```
beatles.catalog %>%  select(-num.tracks)
```

```
## # A tibble: 3 x 2
##   album                  year
##   <chr>                 <dbl>
## 1 Please Please Me      1963
## 2 Rubber Soul           1965
## 3 Magical Mystery Tour  1967
```

Renaming variables can also be done using `select()`. The syntax will be simple: `select(NEW NAME = OLD NAME)`. Example:

```
starwars %>%
  select(alias=name, crib=homeworld, sex=gender)
```

```
## # A tibble: 87 x 3
##    alias              crib     sex
##    <chr>              <chr>    <chr>
##  1 Luke Skywalker     Tatooine masculine
##  2 C-3PO              Tatooine masculine
##  3 R2-D2              Naboo    masculine
##  4 Darth Vader        Tatooine masculine
##  5 Leia Organa        Alderaan feminine
##  6 Owen Lars          Tatooine masculine
##  7 Beru Whitesun lars Tatooine feminine
##  8 R5-D4              Tatooine masculine
##  9 Biggs Darklighter  Tatooine masculine
## 10 Obi-Wan Kenobi     Stewjon  masculine
## # … with 77 more rows
```

Select helpers

- `starts_with()`: Starts with a prefix

Example Select country names and GDP variables from `g3`.

```
g3 %>% select(`Country Name`, starts_with("YR"))
```

```
## # A tibble: 264 x 21
##    `Country Name` YR1999 YR2000 YR2001  YR2002 YR2003 YR2004 YR2005 YR2006
##    <chr>           <dbl>  <dbl>  <dbl>   <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 Afghanistan     NA    NA      NA     NA      8.83   1.41  11.2     5.36
##  2 Albania         12.9   6.95    8.29   4.54   5.53   5.51   5.53    5.90
##  3 Algeria          3.20  3.82    3.01   5.61   7.20   4.30   5.91    1.68
##  4 American Samoa  NA    NA      NA     NA      0.814  0.538 -0.402  -4.17
##  5 Andorra          4.10  3.53    4.55   6.47  12.2    7.65   7.40    4.54
##  6 Angola           2.18  3.05    4.21  13.7    2.99  11.0    15.0    11.5
##  7 Antigua and B~   3.71  6.69   -4.95   1.02   6.06   5.74   6.41   12.7
##  8 Arab World       1.80  5.48    1.61   0.586  5.32   9.34   5.72    6.50
##  9 Argentina       -3.39 -0.789  -4.41 -10.9    8.84   9.03   8.85    8.05
## 10 Armenia          3.30  5.9     9.56  13.2   14.0   10.5    13.9    13.2
## # … with 254 more rows, and 12 more variables: YR2007 <dbl>, YR2008 <dbl>,
## #   YR2009 <dbl>, YR2010 <dbl>, YR2011 <dbl>, YR2012 <dbl>, YR2013 <dbl>,
## #   YR2014 <dbl>, YR2015 <dbl>, YR2016 <dbl>, YR2017 <dbl>, YR2018 <dbl>
```

- `contains()`: Contains a literal string

Example Pick all those variables containing the word `color` from the `starwars` dataset.

```
starwars %>% select(name, contains("color"))
```

```
## # A tibble: 87 x 4
##    name               hair_color    skin_color  eye_color
##    <chr>              <chr>         <chr>       <chr>
##  1 Luke Skywalker     blond         fair        blue
##  2 C-3PO              <NA>          gold        yellow
##  3 R2-D2              <NA>          white, blue red
##  4 Darth Vader        none          white       yellow
##  5 Leia Organa        brown         light       brown
##  6 Owen Lars          brown, grey   light       blue
##  7 Beru Whitesun lars brown         light       blue
##  8 R5-D4              <NA>          white, red  red
##  9 Biggs Darklighter  black         light       brown
## 10 Obi-Wan Kenobi     auburn, white fair        blue-gray
## # … with 77 more rows
```

- `num_range()`: Matches a numerical range like x01, x02, x03

Example Select GDP data and country names from g3 during 2005 and 2010.

```r
g3 %>% select(`Country Name`, num_range("YR",2005:2010))
```

```
## # A tibble: 264 x 7
##    `Country Name`      YR2005 YR2006  YR2007  YR2008  YR2009 YR2010
##    <chr>                <dbl>  <dbl>   <dbl>   <dbl>   <dbl>  <dbl>
##  1 Afghanistan         11.2      5.36 13.8      3.92    21.4   14.4
##  2 Albania              5.53     5.90  5.98     7.50     3.35   3.71
##  3 Algeria              5.91     1.68  3.37     2.36     1.63   3.63
##  4 American Samoa      -0.402  -4.17  1.96    -2.61    -4.24   0.442
##  5 Andorra              7.40     4.54  0.0400 -8.59    -3.69  -5.36
##  6 Angola              15.0     11.5  14.0    11.2      0.859  4.86
##  7 Antigua and Barbuda  6.41    12.7   9.26   -0.0301 -12.1   -7.20
##  8 Arab World           5.72     6.50  4.57     5.82     0.428  4.77
##  9 Argentina            8.85     8.05  9.01     4.06    -5.92  10.1
## 10 Armenia             13.9     13.2  13.7      6.90   -14.1    2.20
## # … with 254 more rows
```

- `ends_with()`: Ends with a suffix
- `one_of()`: Matches variable names in a character vector
- `everything()`: Matches all variables
- `last_col()`: Select last variable, possibly with an offset
- `matches()`: Matches a regular expression (a sequence of symbols/characters expressing a string/pattern to be searched for within text)

`relocate()`

`relocate()` helps you organize columns by changing column positions.

Example Take `beatles.catalog`. Reorder columns such that `year` appears first.

```r
beatles.catalog %>% relocate(year)
```

```
## # A tibble: 3 x 3
##    year album                num.tracks
##   <dbl> <chr>                     <dbl>
## 1  1963 Please Please Me             14
## 2  1965 Rubber Soul                  14
## 3  1967 Magical Mystery Tour         11
```

You can also reorder columns by their types. For example, if you wish to organize `beatles.catalog` such that numeric objects appear first followed by character, you can do this by writing: `beatles.catalog %>% relocate(where(is.numeric))`

`summarize()`

`summarize()`summarizes variables—you choose the variables and the summaries (*e.g.*, `mean()` or `min()`).

```
df %>% summarize(
  mean(x), mean(y),
  min(x), max(x),
  min(y), max(y)
)
```

```
##   mean(x) mean(y) min(x) max(x) min(y) max(y)
## 1      12      11      2     22      1     20
```

returns a 1×6 tibble with the means of `x`, `y`; the minimum of `x` and `y`; and the maximum of `x` and `y`.

`summarize()` and `group_by()`

`group_by()` groups your observations by the variable(s) that you name.

Specifically, `group_by()` returns a *grouped data frame* that you can then feed to `summarize()`, `mutate()` to perform grouped calculations, *e.g.*, each group's mean.

Example: Grouped summaries

```
# Create a new data frame
our_df <- tibble(
  df,
  grp = rep(c("A", "B"), each = 3)
)
```

```
## # A tibble: 6 x 3
##       x     y grp
##   <dbl> <int> <chr>
## 1     2    20 A
## 2     6     1 A
## 3    10    10 A
## 4    14     5 B
## 5    18    14 B
## 6    22    16 B
```

```
# For dataset 'our_df' ...
our_df %>%
  # Group by 'grp'
  group_by(grp) %>%
  # Take means of 'x' and 'y'
  summarize(mean(x), mean(y))
```

```
##   mean(x) mean(y)
## 1      12      11
```

Example: Grouped mutation

```
# Create a new data frame
our_df <- data.frame(
    df,
    grp = rep(c("A", "B"), each = 3)
)
```

```
##    x  y grp
## 1  2 20   A
## 2  6  1   A
## 3 10 10   A
## 4 14  5   B
## 5 18 14   B
## 6 22 16   B
```

```
# Add grp means for x and y
our_df %>%
  group_by(grp) %>%
  mutate(
    x_m = mean(x), y_m = mean(y)
  )
```

```
## # A tibble: 6 x 5
## # Groups:   grp [2]
##       x     y grp     x_m   y_m
##   <dbl> <int> <chr> <dbl> <dbl>
## 1     2    20 A        12    11
## 2     6     1 A        12    11
## 3    10    10 A        12    11
## 4    14     5 B        12    11
## 5    18    14 B        12    11
## 6    22    16 B        12    11
```

`filter()`

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

Example:

```
# Create a dataset
some_df <- data.frame(
  x = 1:10,
  y = 11:20
)
```

```
# Only keep rows where x is 3
some_df %>% filter(x == 3)
```

```
##   x  y
## 1 3 13
```

Using the same dataset and `filter`, perform the following operations-

- keep rows where `x >= 6`
- keep rows where `y/x >= 2`
- keep rows where `12 <= y <= 18`

`arrange()`

`arrange()` will sort the rows of a data frame using the inputted columns.

`R` defaults to starting with the "lowest" (smallest) at the top of the data frame. Use a `-` in front of the variable's name to reverse sort.

```
# As is
our_df
```

```
##    x  y grp
## 1  2 20   A
## 2  6  1   A
## 3 10 10   A
## 4 14  5   B
## 5 18 14   B
## 6 22 16   B
```

```
# Arrang by y, grp, then -x
our_df %>% arrange(y, grp, -x)
```

```
##    x  y grp
## 1  6  1   A
## 2 14  5   B
## 3 10 10   A
## 4 18 14   B
## 5 22 16   B
## 6  2 20   A
```

`slice()`

`arrange()` will subset the data frame using the row index provided by you.

```
n_rows <- 12:18
slice(mtcars, n_rows)
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
```

`distinct()`

`distinct()` will remove duplicates from your data.

```
name <- c("Siddharth", "Rajshree", "Ankitha", "Ankitha")
CGPA <- c(3.9, 3.5, 3.4, 3.4)
age <- c(21, 22, 24, 24)
school.db <- data.frame(name, CGPA, age)
```

```
print(school.db)
```

```
##        name CGPA age
## 1 Siddharth  3.9  21
## 2  Rajshree  3.5  22
## 3   Ankitha  3.4  24
## 4   Ankitha  3.4  24
```

```
school.db %>% distinct()
```

```
##        name CGPA age
## 1 Siddharth  3.9  21
## 2  Rajshree  3.5  22
## 3   Ankitha  3.4  24
```

Chain Operation Revisited

Let's combine several `dplyr` operations into a chain.

Select data → Select groups → Select columns → Compute averages for selected columns

In this example, we will calculate average height and mass by species and sex.

```r
starwars %>%                              #select data
  group_by(species, sex) %>%              #group variables
  select(height, mass) %>%                #select columns
  dplyr::summarise(                       #compute averages
    ave.height = mean(height, na.rm = TRUE),
    ave.mass = mean(mass, na.rm = TRUE)
  )
```

```
## Adding missing grouping variables: `species`, `sex`
```

```
## `summarise()` regrouping output by 'species' (override with `.groups` argument)
```

```
## # A tibble: 41 x 4
## # Groups:   species [38]
##    species   sex    ave.height ave.mass
##    <chr>     <chr>       <dbl>    <dbl>
##  1 Aleena    male          79      15  
##  2 Besalisk  male         198     102  
##  3 Cerean    male         198      82  
##  4 Chagrian  male         196     NaN  
##  5 Clawdite  female       168      55  
##  6 Droid     none         131.     69.8
##  7 Dug       male         112      40  
##  8 Ewok      male          88      20  
##  9 Geonosian male         183      80  
## 10 Gungan    male         209.     74  
## # … with 31 more rows
```

Done for the day

```
## Sorry, this silly GIF is only available in the the HTML version of the notes.
```