SM-2302 Software for Mathematicians

R3: The `tidyverse`

Dr. Haziq Jamil
Mathematical Sciences, Faculty of Science, UBD
https://haziqj.ml

Semester I 2023/24

Preamble

```r
# If not installed yet, install them first
library(tidyverse)
library(remotes)

# This may require compilation
remotes::install_github("rstudio/EDAWR")
library(EDAWR)  # to get the data sets: storms, cases, pollution, tb
```

Note that installing {EDAWR} package may require compilation. For Windows, check out Rtools42. For Macs, I think minimally you will need to install Xcode. Check out this link.

These slides were adapted from the following YouTube playlist.

- Data wrangling with R and the Tidyverse by Garrett Grolemund, RStudio.

I would recommend that you watch them, but note that the video is based on the old version of `tidyverse` packages, so some commands may be deprecated or superseded.

Learning objectives

- Spot the **variables** and **observations** within your data
  - Variables = Columns
  - Observations = Rows
- **Reshape** your data into the layout that works best for R
  - Long vs wide data sets
  - Long → wide using `pivot_wider()`
  - Wide → long using `pivot_longer()`
- *Quickly* derive new variables and observations to explore
  - `mutate()` or `summarise()` to add new variables or summarise data
  - `select()`, `filter()`, `slice()`, `arrange()` to focus and/or reveal information
- Perform **group-wise** summaries to explore hidden levels of information within your data
  - `group_by()`
- Join multiple data sets together
  - `bind_cols()`, `bind_rows()`, `left_join()`, `*_join()`

The `tidyverse` package

The `tidyverse` is a collection of R packages designed for data science. Today we're most interested in the following packages:

1. `tibble`
   A modern re-imagining of `data.frames`.
2. `tidyr`
   Provides a set of functions that help you get to tidy data.
3. `dplyr`
   Provides a set of verbs for data manipulation.

Tibbles: Modern data frames

```
library(tibble)
```

```
iris
```

```
##     Sepal.Length Sepal.Width Petal.Length
## 1            5.1         3.5          1.4
## 2            4.9         3.0          1.4
## 3            4.7         3.2          1.3
## 4            4.6         3.1          1.5
## 5            5.0         3.6          1.4
## 6            5.4         3.9          1.7
## 7            4.6         3.4          1.4
## 8            5.0         3.4          1.5
## 9            4.4         2.9          1.4
## 10           4.9         3.1          1.5
## 11           5.4         3.7          1.5
## 12           4.8         3.4          1.6
## 13           4.8         3.0          1.4
## 14           4.3         3.0          1.1
## 15           5.8         4.0          1.2
## 5.7          4.4
```

```
(tbl_iris <- as_tibble(iris))
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Peta
##           <dbl>       <dbl>        <dbl>
##  1          5.1         3.5          1.4
##  2          4.9         3            1.4
##  3          4.7         3.2          1.3
##  4          4.6         3.1          1.5
##  5          5           3.6          1.4
##  6          5.4         3.9          1.7
##  7          4.6         3.4          1.4
##  8          5           3.4          1.5
##  9          4.4         2.9          1.4
## 10          4.9         3.1          1.5
## # i 140 more rows
```

Tibbles: Modern data frames (cont.)

Some features of `tibbles`:

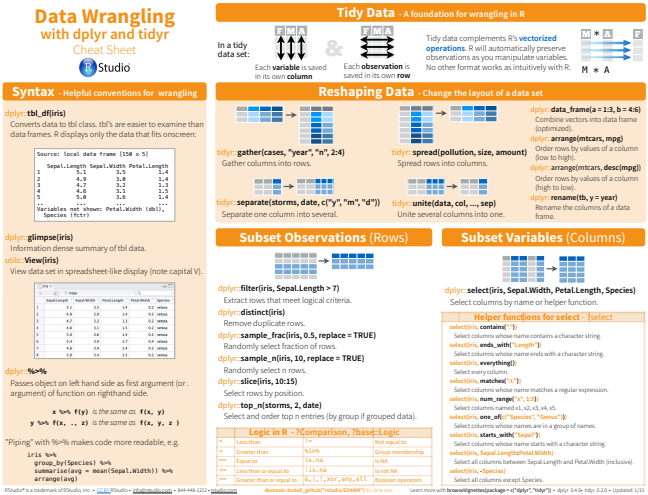
1. An ehanced `print()` method, making it easier to use with large data sets containing complex objects.
2. They are lazy
   - Subsetting results in another `tibble` (does not `drop` dimensions).
   - Partial matching does not work–you need to type the full variable name.
   - `stringsAsFactors = FALSE` by default (for character data).
   - Only vectors of length 1 will undergo length coercion.
3. They are surly (complains more!)

- Forces you to confront problems earlier, leading to cleaner and more expressive code.

Additional reading: `https://r4ds.had.co.nz/tibbles.html`

TL;DR–Use the cheat sheet!

Have a look at the Data Wrangling cheat sheet from RStudio.



Data Wrangling
with dplyr and tidyr
Cheat Sheet
RStudio

Syntax - Helpful conventions for wrangling

dplyr::**tbl_df(iris)**
Converts data to tbl class. tbl's are easier to examine than data frames. R displays only the data that fits onscreen:

```
Source: local data frame [150 x 5]

   Sepal.Length Sepal.Width Petal.Length
1           5.1         3.5          1.4
2           4.9         3.0          1.4
3           4.7         3.2          1.3
4           4.6         3.1          1.5
5           5.0         3.6          1.4
..          ...         ...          ...
Variables not shown: Petal.Width (dbl),
  Species (fctr)
```

dplyr::**glimpse(iris)**
Information dense summary of tbl data.

utils::**View(iris)**
View data set in spreadsheet-like display (note capital V).

dplyr::**%>%**
Passes object on left hand side as first argument (or . argument) of function on righthand side.

`x %>% f(y)` *is the same as* `f(x, y)`
`y %>% f(x, .., z)` *is the same as* `f(x, y, z )`

"Piping" with %>% makes code more readable, e.g.

```
iris %>%
  group_by(Species) %>%
  summarise(avg = mean(Sepal.Width)) %>%
  arrange(avg)
```

Tidy Data - A foundation for wrangling in R

In a tidy data set:
Each **variable** is saved in its own **column**
Each **observation** is saved in its own **row**

Tidy data complements R's **vectorized operations**. R will automatically preserve observations as you manipulate variables. No other format works as intuitively with R.

M * A

Reshaping Data - Change the layout of a data set

tidyr::**gather(cases, "year", "n", 2:4)**
Gather columns into rows.

tidyr::**spread(pollution, size, amount)**
Spread rows into columns.

tidyr::**separate(storms, date, c("y", "m", "d"))**
Separate one column into several.

tidyr::**unite(data, col, ..., sep)**
Unite several columns into one.

dplyr::**data_frame(a = 1:3, b = 4:6)**
Combine vectors into data frame (optimized).

dplyr::**arrange(mtcars, mpg)**
Order rows by values of a column (low to high).

dplyr::**arrange(mtcars, desc(mpg))**
Order rows by values of a column (high to low).

dplyr::**rename(tb, y = year)**
Rename the columns of a data frame.

Subset Observations (Rows)

dplyr::**filter(iris, Sepal.Length > 7)**
Extract rows that meet logical criteria.

dplyr::**distinct(iris)**
Remove duplicate rows.

dplyr::**sample_frac(iris, 0.5, replace = TRUE)**
Randomly select fraction of rows.

dplyr::**sample_n(iris, 10, replace = TRUE)**
Randomly select n rows.

dplyr::**slice(iris, 10:15)**
Select rows by position.

dplyr::**top_n(storms, 2, date)**
Select and order top n entries (by group if grouped data).

Logic in R – ?Comparison, ?base::Logic			
<	Less than	!=	Not equal to
>	Greater than	%in%	Group membership
==	Equal to	is.na	Is NA
<=	Less than or equal to	!is.na	Is not NA
>=	Greater than or equal to	&, |, !, xor, any, all	Boolean operators

Subset Variables (Columns)

dplyr::**select(iris, Sepal.Width, Petal.Length, Species)**
Select columns by name or helper function.

Select helper functions for select - ?select	
select(iris, **contains**("."))	Select columns whose name contains a character string.
select(iris, **ends_with**("Length"))	Select columns whose name ends with a character string.
select(iris, **everything**())	Select every column.
select(iris, **matches**("t."))	Select columns whose name matches a regular expression.
select(iris, **num_range**("x", 1:3))	Select columns named x1, x2, x3, x4, x5.
select(iris, **one_of**(c("Species", "Genus")))	Select columns whose names are in a group of names.
select(iris, **starts_with**("Sepal"))	Select columns whose name starts with a character string.
select(iris, Sepal.Length:Petal.Width)	Select all columns between Sepal.Length and Petal.Width (inclusive).
select(iris, -Species)	Select all columns except Species.

RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com

devtools::install_github("rstudio/EDAWR") for data sets

Learn more with browseVignettes(package = c("dplyr", "tidyr")) • dplyr 0.4.0• tidyr 0.2.0 • Updated 1/15

Tidy data

Data comes in various shapes and formats. It's important when you begin your analysis to identify this at the outset.

- What are the variables? How many of them are there?
- What are the observations? How many of them are there?

The goal is to reshape your data into a format that works best in R. The resulting format is called **tidy data**.

Let's have a look at three data sets:

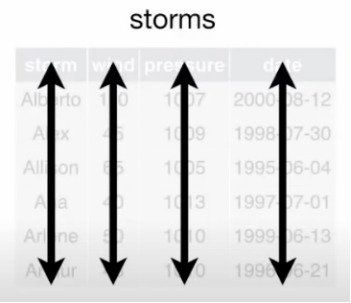
1. `storms` (Wind speed data for six hurricanes)
2. `cases` (Data from the WHO Global Tuberculosis report)
3. `pollution` (Ambient air polution from WHO)

1. `storms` data set

```
EDAWR::storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

- Variables: `storm`, `wind`, `pressure`, `date`
- Observations: 1, 2, 3, 4, 5, 6
- Notice that
  - Each variable is represented by a single column
  - Each row is a single observation



Extracting data by subsetting easily:

- `storms$storm`
- `storms$wind`
- `storms$pressure`
- `storms$date`

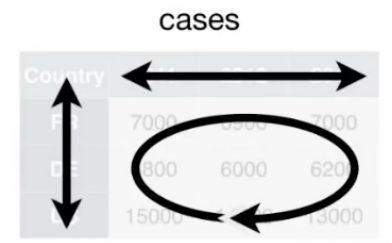
2. cases data set

Wide format

```
EDAWR::cases
```

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

- Variables: `country`, `year = c(2011, 2012, 2013)`, `count`
- Observations: $3 \times 3 = 9$
- Notice that
  - Each cell of this data frame corresponds to the `count` for a given `country` and `year`
  - Different from the `storms` data set!



Extracting data by subsetting (not so easy):

- `cases$country`
- `colnames(cases)[-1]`
- `unlist(cases[1:3, 2:4]`

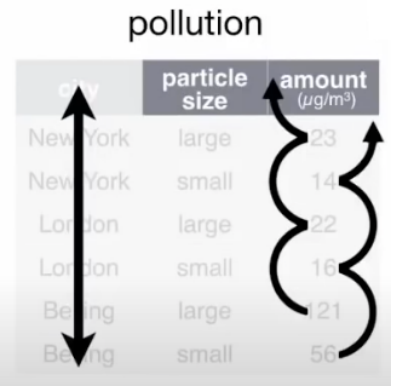
3. `pollution` data set

Long format

```
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
```

- Variables: `city`, `amount`
- Observations: 1, 2, 3, 4, 5, 6
- Notice that `amount` for each city is segregated into two groups: `large` and `small`.



Extracting data by subsetting (hard):

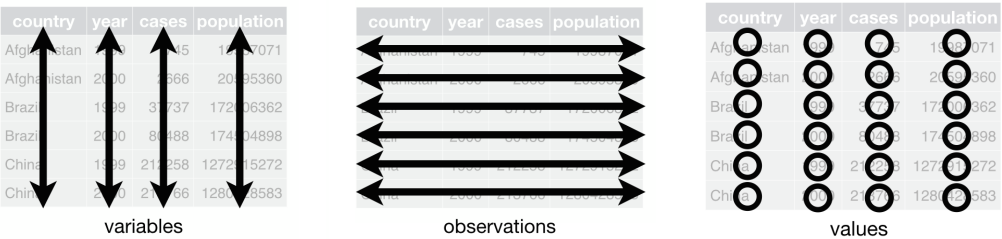
- `pollution$city[c(1, 3, 5)]`
- `pollution$amount[c(1, 3, 5)]`
- `pollution$amount[c(2, 4, 6)]`

Tidy data

The defining characteristics of tidy data are:

> *Variables in columns, observations in rows, and each type in a table.*

This makes variables <u>easy to access</u> and manipulate (while preserving observations).



Tidy data

Getting tidy data (reshaping)

- `pivot_longer()`
- `pivot_wider()`
- `separate()`
- `unite()`
- `arrange()`
- `rename()`

Data wrangling

Pipeline

Group wrangling

Joining data sets

Getting tidy data (reshaping)

There are two main functions that we will use to reshape the layout of tables:

1. `pivot_wider()`
2. `pivot_longer()`

To a lesser degree, these utility functions may be useful too:

1. `separate()`
2. `unite()`
3. `arrange()`
4. `rename()`

pivot_longer()

This converts data from wide format to long format. In the cases data set, ideally we want to have three columns only: `country`, `year`, and `count`; and each row will be an observation.

```
cols = c("2011",
         "2012",
         "2013")
```

country	2011	2012	2013
FR	7000	6900	7000
DE	5800	6000	6200
US	15000	14000	13000

pivot_longer()

```
values_to = "count"
names_to = "year"
```

country	year	count
FR	2011	7000
FR	2012	6900
FR	2013	7000
DE	2011	5800
DE	2012	6000
DE	2013	6200
US	2011	15000
US	2012	14000
US	2013	13000

pivot_longer() (cont.)

```
cases
```

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

- Select which `cols` to pivot to longer format. Note that these have to be character vectors.
- `names_to` is the name of the new column to store the old columns
- `values_to` is the name of the new column to store the observations

```
pivot_longer(data = cases,
             cols = c("2011", "2012", "2013"),
             names_to = "year",
             values_to = "count")
```

```
## # A tibble: 9 x 3
##   country year  count
##   <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
```

pivot_wider()

This converts data from long format to wide format. In the `pollution` data set, we could instead have a table of city by particle size, and each cell is the amount.

`id_cols = "city"`

`names_from = "size"`

`values_from = "amount"`

city	size	amount
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

`pivot_wider()`

city	large	small
New York	23	14
London	22	16
Beijing	121	56

pivot_wider() (cont.)

```r
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
```

- Select which `id_cols` uniquely identifies each observation
- `names_from` is the name of the column to spread
- `values_from` is the name of column containing the observations

```r
pivot_wider(data = pollution,
            id_cols = "city",
            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
```

separate()

Turns a single character column into multiple columns.

```
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```
separate(data = storms,
         col = "date",
         into = c("year", "month", "day"),
         sep = "-")
```

```
## # A tibble: 6 x 6
##   storm    wind pressure year  month day
##   <chr>   <int>    <int> <chr> <chr> <chr>
## 1 Alberto   110     1007 2000  08    03
## 2 Alex       45     1009 1998  07    27
## 3 Allison    65     1005 1995  06    03
## 4 Ana        40     1013 1997  06    30
## 5 Arlene     50     1010 1999  06    11
## 6 Arthur     45     1010 1996  06    17
```

unite()

Paste together multiple columns into one.

```r
storms2
```

```
## # A tibble: 6 x 6
##   storm    wind pressure year  month day
##   <chr>   <int>    <int> <chr> <chr> <chr>
## 1 Alberto   110     1007 2000  08    03
## 2 Alex       45     1009 1998  07    27
## 3 Allison    65     1005 1995  06    03
## 4 Ana        40     1013 1997  06    30
## 5 Arlene     50     1010 1999  06    11
## 6 Arthur     45     1010 1996  06    17
```

```r
unite(data = storms2,
      col = "date",
      "year", "month", "day",
      sep = "-")
```

```
## # A tibble: 6 x 4
##   storm    wind pressure date
##   <chr>   <int>    <int> <chr>
## 1 Alberto   110     1007 2000-08-03
## 2 Alex       45     1009 1998-07-27
## 3 Allison    65     1005 1995-06-03
## 4 Ana        40     1013 1997-06-30
## 5 Arlene     50     1010 1999-06-11
## 6 Arthur     45     1010 1996-06-17
```

arrange()

Sorting rows (in ascending order) in a particular column is done using `arrange()`.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
arrange(.data = storms, wind)
```

```
##     storm wind pressure       date
## 1     Ana   40     1013 1997-06-30
## 2    Alex   45     1009 1998-07-27
## 3  Arthur   45     1010 1996-06-17
## 4  Arlene   50     1010 1999-06-11
## 5 Allison   65     1005 1995-06-03
## 6 Alberto  110     1007 2000-08-03
```

arrange() (cont.)

Sorting rows (in descending order) in a particular column is done by applying desc() on the variable.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
arrange(.data = storms, desc(wind))
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2 Allison   65     1005 1995-06-03
## 3  Arlene   50     1010 1999-06-11
## 4    Alex   45     1009 1998-07-27
## 5  Arthur   45     1010 1996-06-17
## 6     Ana   40     1013 1997-06-30
```

arrange() (cont.)

You may arrange by multiple columns in order.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
arrange(.data = storms, wind, date)
```

```
##     storm wind pressure       date
## 1     Ana   40     1013 1997-06-30
## 2  Arthur   45     1010 1996-06-17
## 3    Alex   45     1009 1998-07-27
## 4  Arlene   50     1010 1999-06-11
## 5 Allison   65     1005 1995-06-03
## 6 Alberto  110     1007 2000-08-03
```

rename()

A utility function to rename the columns.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
rename(.data = storms,
       WIND = wind,
       Storm = storm)
```

```
##     Storm WIND pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

Tidy data

Getting tidy data (reshaping)

Data wrangling

- `select()`
- `filter()`
- `slice()`
- `mutate()`
- `summarise()`

Pipeline

Group wrangling

Joining data sets

Data wrangling

Having obtained a tidy dataset, there are several ways to access information.

1. **Extract** existing variables (columns).
   - `select()`
2. **Extract** existing observations (rows).
   - `filter()` or `slice()`
3. **Derive** new variables (from existing variables).
   - `mutate()`
4. **Change** the unit of analysis.
   - `summarise()`

Notice that these functions are *verbs* (action words). There are many more, of course. Read more: `https://dplyr.tidyverse.org/articles/programming.html`

select()

Select variables in a data frame.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
select(.data = storms, storm, pressure)
```

```
##     storm pressure
## 1 Alberto     1007
## 2    Alex     1009
## 3 Allison     1005
## 4     Ana     1013
## 5  Arlene     1010
## 6  Arthur     1010
```

Remark

You do not need to use quotes when choosing the variable names! I.e. `select(storms, "storm", "pressure")` is not necessary.

select(): Optionally rename

Optionally rename while selecting.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
select(.data = storms, STORM = storm,
       PRESSURE = pressure)
```

```
##     STORM PRESSURE
## 1 Alberto     1007
## 2    Alex     1009
## 3 Allison     1005
## 4     Ana     1013
## 5  Arlene     1010
## 6  Arthur     1010
```

select(): Negative subsetting

Deselecting variables.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
select(.data = storms, -storm)
```

```
##   wind pressure       date
## 1  110     1007 2000-08-03
## 2   45     1009 1998-07-27
## 3   65     1005 1995-06-03
## 4   40     1013 1997-06-30
## 5   50     1010 1999-06-11
## 6   45     1010 1996-06-17
```

`select()`: Selecting a range of consecutive variables

Use `:` to select a range of consecutive variables

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
select(.data = storms, storm:pressure)
```

```
##     storm wind pressure
## 1 Alberto  110     1007
## 2    Alex   45     1009
## 3 Allison   65     1005
## 4     Ana   40     1013
## 5  Arlene   50     1010
## 6  Arthur   45     1010
```

Useful select functions

Call	Use
`-`	Select everything but
`:`	Select range
`contains()`	Select columns whose name contains a character string
`starts_with()`	Select column whose name starts with a character string
`ends_with()`	Select columns whose nmame ends with a string
`everything()`	Select all columns
`matches()`	Select columns whose name matches a regular expression
`num_range()`	Select columns matching a numerical range e.g. `x1`, `x2`, etc.
`one_of()`	Select columns whose names are in a group of names

There are others. See `?select` for further details.

filter()

This is used to subset a data frame, retaining all rows that satisfy your logical tests.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
filter(.data = storms, wind >= 50)
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2 Allison   65     1005 1995-06-03
## 3  Arlene   50     1010 1999-06-11
```

filter() (cont.)

You can combine tests separated by commas.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
filter(.data = storms,
       wind >= 50,
       storm %in% c("Alberto", "Alex",
                    "Allison"))
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2 Allison   65     1005 1995-06-03
```

Logical tests in R

`?Comparison`

Test	Usage
`<`	Less than
`>`	Greater than
`==`	Equal to
`<=`	Less than or equal to
`>=`	Greater than or equal to
`!=`	Not equal to
`%in%`	Group membership
`is.na()`	Is NA?

`?base::Logic`

Operator	Usage
`&`	Boolean and
`|`	Boolean or
`xor`	Exactly or
`!`	Not
`any()`	Any true
`all()`	All true

slice()

This lets you index rows by their (integer) locations. Thus, it allows you to select, remove, and duplicate rows.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
slice(.data = storms, 1:3)
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
```

```r
slice(.data = storms, rep(1, 3))
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2 Alberto  110     1007 2000-08-03
## 3 Alberto  110     1007 2000-08-03
```

slice() (cont.)

Using `which.min()` or `which.max()` (or other functions which return row indices) is quite helpful with `slice()`.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
slice(.data = storms, which.min(pressure))
```

```
##     storm wind pressure       date
## 1 Allison   65     1005 1995-06-03
```

```r
slice(.data = storms, which.max(wind))
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
```

mutate()

We may want to **create** new variables from existing variables. Suppose we want to derive a new variable called `ratio` which is defined as

$$\texttt{ratio} = \frac{\texttt{pressure}}{\texttt{wind}}$$

storm	wind	pressure	date
Alberto	110	1007	2000-08-03
Alex	45	1009	1998-07-27
Allison	65	1005	1995-06-03
Ana	40	1013	1997-06-30
Arlene	50	1010	1999-06-11
Arthur	45	1010	1999-06-17

storms$pressure	/	storms$wind	→	ratio
1007	/	110	→	9.16
1009	/	45	→	22.42
1005	/	65	→	15.46
1013	/	40	→	25.33
1010	/	50	→	20.20
1010	/	45	→	22.44

mutate() (cont.)

Using dplyr's mutate() function, we are able to do this easily without having to use $ all the time.

```r
# Before
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

```r
# After
mutate(.data = storms,
       ratio = pressure / wind)
```

```
##     storm wind pressure       date     ratio
## 1 Alberto  110     1007 2000-08-03  9.154545
## 2    Alex   45     1009 1998-07-27 22.422222
## 3 Allison   65     1005 1995-06-03 15.461538
## 4     Ana   40     1013 1997-06-30 25.325000
## 5  Arlene   50     1010 1999-06-11 20.200000
## 6  Arthur   45     1010 1996-06-17 22.444444
```

mutate() (cont.)

Keep on adding new columns in the same `mutate()` call. Just separate them by commas.

```
mutate(.data = storms,
       ratio = pressure / wind,
       inverse = ratio ^ (-1))
```

```
##     storm wind pressure       date     ratio    inverse
## 1 Alberto  110     1007 2000-08-03  9.154545 0.10923535
## 2    Alex   45     1009 1998-07-27 22.422222 0.04459861
## 3 Allison   65     1005 1995-06-03 15.461538 0.06467662
## 4     Ana   40     1013 1997-06-30 25.325000 0.03948667
## 5  Arlene   50     1010 1999-06-11 20.200000 0.04950495
## 6  Arthur   45     1010 1996-06-17 22.444444 0.04455446
```

Useful `mutate()` functions

Operator	Usage
`pmin()`, `pmax()`	Element-wise min and max
`cummin()`, `cummax()`	Cumulative min and max
`cumsum()`, `cumprod()`	Cumulative sum and product
`between()`	Are values between `a` and `b`?
`cumall()`, `cumany()`	Cumulative `all()` and `any()`
`cummean()`	Cumulative mean
`lead()`, `lag()`	Comparing values behind or ahead of current values
`ntile()`	Bin vector into `n` buckets
`row_number()`	Returns row number

Remark

All of these (window) functions take vector values and return vector values of the same length. If using non-window functions, the recycling rule applies.

summarise()

On the other hand, we may want to **condense** the available information. For this, the `summarise()` function returns a new data frame.

```r
# Before
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
```

```r
# After
summarise(.data = pollution,
          median = median(amount),
          variance = var(amount))
```

```
##   median variance
## 1   22.5   1731.6
```

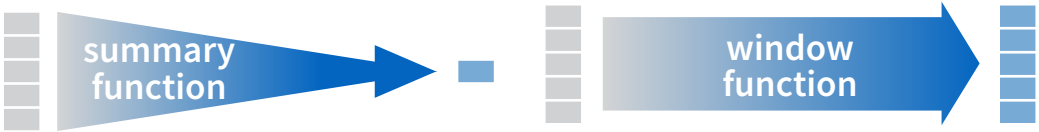
Useful `summarise()` functions

Operator	Usage
`min()`, `max()`	Minimum and maximum values
`mean()`	Mean value
`median()`	Median value
`sum()`	Sum of values
`var()`, `sd()`	Variance and standard deviation of a vector
`first()`, `last()`	First or last value in a vector
`nth()`	Nth value in a vector
`n()`	The number of values in a vector
`n_distinct()`	The number of distinct values in a vector

Remark

All of these functions take vector values and return a single value.

Summary (`summarise`) vs window (`mutate`) functions



- Functions used with `summarise()` should reduce the length of the input vector to a single value.
- Functions used with `mutate()` should keep the vector length.

Tidy data

Getting tidy data (reshaping)

Data wrangling

Pipeline

Group wrangling

Joining data sets

Pipeline

Consider the following sequence of actions that describe the process of getting to UBD campus everyday:

> *I need to find my key, then unlock my car, then start my car, then drive to school, then park.*

Expressed as a set of nested functions in R pseudocode this would look like:

```
park(drive(start_car(find("keys")), to = "campus"))
```

Writing it out using pipes give it a more natural (and easier to read) structure:

```
find("keys") %>%
    start_car() %>%
    drive(to = "campus") %>%
    park()
```

The pipe operator %>%

The pipe operator puts the output of the LHS into the first argument of the function of the RHS.

Therefore, the following code both do the same thing.

```
summarise(storms, mean = mean(wind))
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1  59.2
```

```
storms %>% summarise(., mean = mean(wind))
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1  59.2
```

In fact, we may drop the '`.`' when the situation is obvious, i.e. `storms %>% summarise(mean = mean(wind))` would give the same thing.

Combining `dplyr` functions

For data wrangling, it seems more natural to progressively write code in a pipeline path. As an example, consider the `nycflights13::flights` data set. How many flights to LAX did each of the legacy carriers (AA, UA, DL or US) have in May from JFK, and what was their average duration?

```
library(nycflights13)
filtered_flights <-
  filter(.data = flights,
         origin == "JFK", dest == "LAX", month == 5,
         carrier %in% c("AA", "UA", "DL", "US"))
res <- summarise(.data = filtered_flights,
                 n = n(), avg_dur = mean(air_time, na.rm = TRUE))
res
```

```
## # A tibble: 1 x 2
##       n avg_dur
##   <int>   <dbl>
## 1   685    320.
```

Combining dplyr functions (cont.)

In contrast, we can pipe the entire thing:

```
flights %>%
  # Filter first
  filter(origin == "JFK", dest == "LAX", month == 5,
         carrier %in% c("AA", "UA", "DL", "US")) %>%
  # Then summarise
  summarise(
    n = n(),
    avg_dur = mean(air_time, na.rm = TRUE)
  )
```

```
## # A tibble: 1 x 2
##       n avg_dur
##   <int>   <dbl>
## 1   685    320.
```

Another example

```r
flights %>%
  # Select all variables containing "delay" in their name
  select(contains("delay")) %>%
  # Create a new gain variable
  mutate(gain = arr_delay - dep_delay) %>%
  # Drop all rows with NA in them
  drop_na() %>%
  # Summarise
  summarise(
    min = min(gain),
    max = max(gain),
    mean = mean(gain)
  )
```

```
## # A tibble: 1 x 3
##     min   max  mean
##   <dbl> <dbl> <dbl>
## 1  -109   196 -5.66
```

Tidy data

Getting tidy data (reshaping)

Data wrangling

Pipeline

Group wrangling

Joining data sets

Group wrangling

When we used the `summarise()` function, we were in fact using the **entire data set** to arrive at the summaries.

city	size	amount
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

`summarise()` →

sum	n	mean
252	6	42

e.g. `pollution %>% summarise(sum = sum(amount), n = n(), mean = mean(amount))`.

Group wrangling (cont.)

Grouping the observations by some categorical variable allows us to uncover hidden information lying within the groups. We use `group_by()` to do this.

city	size	amount
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

```
group_by(city) %>%
summarise()
```

sum	n	mean
37	2	18.5
38	2	19
177	2	88.5

Group wrangling (cont.)

The code to obtain the previous grouped summary table is

```
pollution %>%
  group_by(city) %>%
  summarise(
    sum = sum(amount),
    n = n(),
    mean = mean(amount)
  )
```

```
## # A tibble: 3 x 4
##   city       sum     n  mean
##   <chr>    <dbl> <int> <dbl>
## 1 Beijing    177     2  88.5
## 2 London      38     2  19
## 3 New York    37     2  18.5
```

ungroup()

Note that when you group data, this will persist throughout the pipeline. This is indicated by the `Groups: city [3]` print out below. If, further down the pipeline, you wish to mutate or summarise based on the entire data set, you must first `ungroup()`.

```
pollution %>%
  group_by(city)
```

```
## # A tibble: 6 x 3
## # Groups:   city [3]
##   city     size  amount
##   <chr>    <chr>  <dbl>
## 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
```

```
pollution %>%
  group_by(city) %>%
  ungroup()
```

```
## # A tibble: 6 x 3
##   city     size  amount
##   <chr>    <chr>  <dbl>
## 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
```

Multiple groups

Multiple groupings are allowed. Let's find the top 3 routes with the smallest departure delay in the `flights` dataset.

```
flights %>%
  group_by(origin, dest) %>%
  summarise(min_delay = min(dep_delay)) %>%
  arrange(min_delay) %>%
  print(n = 3)
```

```
## # A tibble: 224 x 3
## # Groups:   origin [3]
##   origin dest  min_delay
##   <chr>  <chr>     <dbl>
## 1 LGA    EYW         -18
## 2 JFK    HNL         -16
## 3 JFK    ACK         -13
## # i 221 more rows
```

Tidy data

Getting tidy data (reshaping)

Data wrangling

Pipeline

Group wrangling

Joining data sets

- `bind_cols()`
- `bind_rows()`
- `left_join()`

Joining data sets

Here are the functions useful for combining data sets

1. `bind_cols()`
2. `bind_rows()`
3. `*_join()` commands
4. Other set operation functions such as `union()`, `intersect()`, and `setdiff()`

For more details on these functions, please have a look at `https://github.com/gadenbuie/tidyexplain` – This page contains a helpful information about what the join functions do.

bind_cols()

If you have two or more data frames (or columns) that are meant to go together column-wise, then use `bind_cols()`.

```
x <- select(storms, 1:3)  # First 3 columns of storms
y <- select(storms, date)  # The last column of storms
bind_cols(x, y)
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

bind_rows()

Similarly if you wanted to stack two or more data frameson top of each other, then use `bind_rows()`.

```
x <- storms[1:3, ]  # First 3 rows of storms
y <- storms[4:6, ]  # Last 3 rows of storms
bind_rows(x, y)
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

Remark

`bind_rows()`

`bind_cols()`

Warning

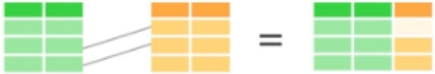
`bind_cols()` and `bind_rows()` have no way of checking whether or not the joining is consistent (e.g. is the row-ordering the same between both data frames in `bind_cols()`?)

left_join()

Out of all the `*_join()` functions, this is probably the most frequently used (at least for me anyway). When calling `left_join(x, y)`, this

- adds columns from `y` to `x`;
- matching rows based on the `by` keys;
- and **includes all rows in `x`** (but possibly not `y`).

left_join()



left_join() (cont.)

An example of left_join() on the band members and instruments data sets.

```
band_members
```

```
## # A tibble: 3 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

```
band_instruments
```

```
## # A tibble: 3 x 2
##   name  plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
```

```
band_members %>%
  left_join(., band_instruments, by = "name")
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```