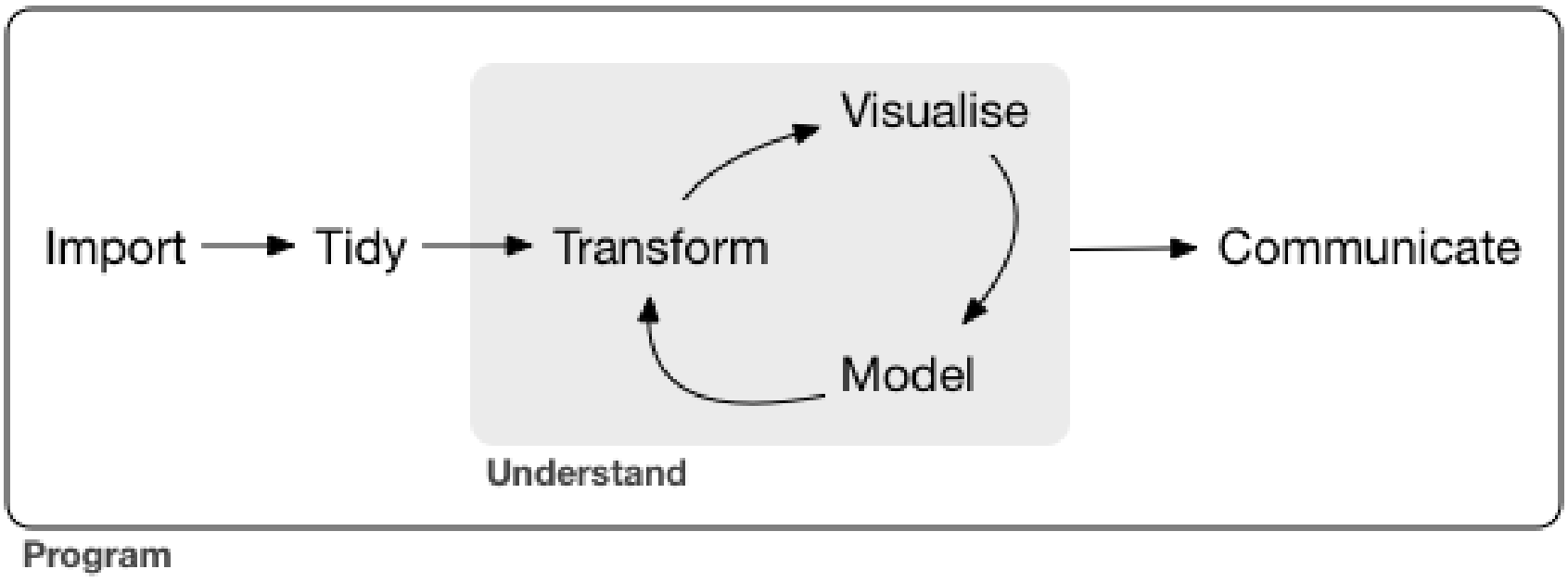
Handling Data - Data Wrangling

Guy J. Abel

Tidyverse

- Some of the functions in `base` can be un-intuitive and slow when dealing with big data sets.
  - For a long time manipulations of data frames in R was done with functions like `with`, `subset`, `merge`, `transform`, `split`, `reshape`.
  - Some of these are really painful to understand (check out `?reshape`)
  - They were also fairly slow and prone to crashing R if you used the wrong values.
- The tidyverse collection of packages are becoming the preferred way of have handling data in R.
- The packages in the tidyverse share a common philosophy of data and R programming, and are designed to work together naturally.
- Tidyverse packages were created in recent years by Hadley Wickham (Chief Scientist at RStudio) and colleagues.

Tidyverse


Visualise
Import
Tidy
Transform
Communicate
Model
Understand
Program

Tidyverse

- The `tidyverse` package itself does not have any functions for importing, manipulating or visualizing data.
- Instead, the `tidyverse` is a wrapper to load eight packages for these purposes
  - `ggplot2` - Create Elegant Data Visualizations Using the Grammar of Graphics
  - `purrr` - Functional Programming Tools
  - `tibble` - Simple Data Frames
  - `dplyr` - A Grammar of Data Manipulation
  - `tidyr` - Easily Tidy Data
  - `stringr` - Simple, Consistent Wrappers for Common String Operations
  - `readr` - Read Rectangular Text Data
  - `forcats` - Tools for Working with Categorical Variables (Factors)

Tidyverse

```
> library(tidyverse)
-- Attaching packages ---------------------------------------- tidyverse 1.3.
v ggplot2 3.3.2     v purrr   0.3.4
v tibble  3.0.3     v dplyr   1.0.1
v tidyr   1.1.1     v stringr 1.4.0
v readr   1.3.1     v forcats 0.5.0
-- Conflicts ------------------------------------------- tidyverse_conflicts(
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

- When the tidyverse package is installed, so are other related packages, such as `readxl` and `haven` that are less frequently required
- These are not loaded with the command `library(tidyverse)`. Needed to be loaded separately.

Tidy Data

- There are many ways to organize data frames.
- However, R likes data if
  - Variables in the data set have their own column
  - Observation are placed in its own row
  - Values are placed in its own cell
- This is what Wickham calls "tidy" data.
- Most functions in R (and Stata, SPSS and SAS) work with data in this format.
  - R is a vectorized programming language, the machinery behind R is optimized to work with vectors.
  - When data is in tidy format R coding is more straightforward and data manipulations are faster.

Tidy Data Philosophy

- Raw data is rarely in a tidy format.
- Tidying data can take much longer than the analysis.
- Learning the grammar of data manipulations to format data to a tidy format.
  - Saves a lot of time.
  - Reduces the steps in managing non-tidy data. These tend to multiply quickly for larger data sets.
  - Reduces the risk of making an errors (through less steps).
  - Provides a consistency to analysis, making extensions far easier.

Reshaping Data

- The `tidyr` package helps get data into a tidy format. At its heart is two functions, that can alter the layout of data sets:
  - `pivot_longer()`
  - `pivot_wider()`.

pivot_longer(data, cols, names_to = "name", values_to = "value"**)**

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

key value

pivot_longer(data = table4a, cols = c("1999", "2000")
names_to = "year", values_to = "cases")

pivot_wider(data, names_from, values_from**)**

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

key value

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

pivot_wider(data = table2, names_from = "type",
values_from = "count")

Pivot Longer

- The `pivot_longer()` function takes values over multiple columns and gathering them to form a database with fewer columns.
- It to has four main arguments for
  - `data`: your data frame.
  - `cols`: variable names (or column numbers) of columns to be stacked
  - `names_to`: the name of a single new column from the column names of `cols` input
  - `value_to`: the name of a single new column from the data stored in cell values.

Pivot Longer

- To demonstrate we will use data from the UN on past total fertility rates (TFR) in all countries.
- Inspect the `ESTIMATES` sheet of `WPP2019_FERT_F04_TOTAL_FERTILITY.xlsx` file.
  - What shall we call `value_to` (describing the values in the cells)?
  - What shall we call `names_to` (describing the column names)?

```
file.show("../data/WPP2019_FERT_F04_TOTAL_FERTILITY.xlsx")
```

```
> df1
# A tibble: 227 x 16
   name    code `1950-1955` `1955-1960` `1960-1965` `1965-1970` `1970-1975`
   <chr> <dbl>       <dbl>       <dbl>       <dbl>       <dbl>       <dbl>
 1 Afri~   903        6.57        6.62        6.70        6.71        6.70
 2 Asia    935        5.83        5.59        5.80        5.75        5.06
 3 Euro~   908        2.66        2.66        2.57        2.37        2.17
 4 Lati~   904        5.83        5.85        5.83        5.46        4.92
 5 Nort~   905        3.34        3.61        3.28        2.55        2.02
 6 Ocea~   909        3.89        4.10        3.97        3.59        3.25
 7 East~   910        6.99        6.99        7.08        7.10        7.14
 8 Buru~   108        6.80        6.86        7.05        7.23        7.26
 9 Como~   174        6           6.60        6.91        7.05        7.05
10 Djib~   262        6.31        6.39        6.55        6.71        6.84
# ... with 217 more rows, and 9 more variables: `1975-1980` <dbl>,
```

Pivot Longer

```
> library(tidyverse)
> # stack data in columns 3 to 16
> df2 <- pivot_longer(data = df1, cols = 3:16,
+                     names_to = "period", values_to = "tfr")
> df2
# A tibble: 3,178 x 4
   name    code period      tfr
   <chr>  <dbl> <chr>     <dbl>
 1 Africa   903 1950-1955  6.57
 2 Africa   903 1955-1960  6.62
 3 Africa   903 1960-1965  6.70
 4 Africa   903 1965-1970  6.71
 5 Africa   903 1970-1975  6.70
 6 Africa   903 1975-1980  6.64
 7 Africa   903 1980-1985  6.50
 8 Africa   903 1985-1990  6.19
 9 Africa   903 1990-1995  5.72
10 Africa   903 1995-2000  5.35
# ... with 3,168 more rows
```

Pivot Longer

- You can use a variety of ways to set `cols` including some helper functions.
- These all give the same data set as before

```
> # from column `1950-1955` to column `2010-2015` (and every column inbetween)
> df2 <- pivot_longer(data = df1, cols = "1950-1955":"2015-2020",
+                     names_to = "period", values_to = "tfr")
> # columns containing "-" in their name
> df2 <- pivot_longer(data = df1, cols = contains("-"),
+                     names_to = "period", values_to = "tfr")
> # without first and second columns
> df2 <- pivot_longer(data = df1, cols = -(1:2),
+                     names_to = "period", values_to = "tfr")
> # without the columns from name to column code (and every column inbetween)
> df2 <- pivot_longer(data = df1, cols = -("name":"code"),
+                     names_to = "period", values_to = "tfr")
```

Pivot Wider

- Where `pivot_longer()` collects values from many columns to one, `pivot_wider()` does the opposite, taking values in one columns and spreading them to form a database with more columns, based on the variables in the data.
- `pivot_wider()` will take excessively long data sets with too many rows and widen them towards a tidy format
- It has three principle arguments (similar to `pivot_longer()`)
  - `data`: your data frame.
  - `names_from`: the name of the column whose values will be used as column names
  - `values_from`: the name of the column whose values will populate the cells.

Pivot Wider

- Most data in the wild does not need to spread.
- The `pivot_wider()` function might be useful when building table based on subset of larger data sets
  - e.g. countries and periods where TFR is less than 1.2

```
> # we will cover filter() soon
> df3 <- filter(df2, tfr < 1.2)
> df3
# A tibble: 17 x 4
   name                                code period      tfr
   <chr>                              <dbl> <chr>     <dbl>
 1 China, Hong Kong SAR                 344 1995-2000  1.06
 2 China, Hong Kong SAR                 344 2000-2005  0.95
 3 China, Hong Kong SAR                 344 2005-2010  1.04
 4 China, Macao SAR                     446 1995-2000  1.12
 5 China, Macao SAR                     446 2000-2005  0.85
 6 China, Macao SAR                     446 2005-2010  0.98
 7 China, Macao SAR                     446 2010-2015  1.19
 8 China, Taiwan Province of China      158 2005-2010  1.05
 9 China, Taiwan Province of China      158 2010-2015  1.11
10 China, Taiwan Province of China      158 2015-2020  1.15
11 Republic of Korea                    410 2005-2010  1.17
12 Republic of Korea                    410 2015-2020  1.11
13 Czechia                              203 1995-2000  1.17
```

Pivot Wider

```
> df4 <- pivot_wider(data = df3, names_from = period, values_from = tfr)
> df4
# A tibble: 8 x 7
  name                code `1995-2000` `2000-2005` `2005-2010` `2010-2015` `2015-2020
  <chr>              <dbl>       <dbl>       <dbl>       <dbl>       <dbl>       <dbl
1 China, Hong Kon~   344         1.06        0.95        1.04        NA          NA
2 China, Macao SAR   446         1.12        0.85        0.98        1.19        NA
3 China, Taiwan P~   158        NA          NA           1.05        1.11         1.1
4 Republic of Kor~   410        NA          NA           1.17        NA           1.1
5 Czechia            203         1.17        1.19       NA          NA          NA
6 Ukraine            804        NA           1.15       NA          NA          NA
7 Latvia             428         1.17       NA          NA          NA          NA
8 Spain              724         1.19       NA          NA          NA          NA
```

Exercise 1 (ex41.R)

```
# 0.  a) Check your working directory is in the course folder. Load the .Rproj file
getwd()
#     b) Load the tidyverse package
library(tidyverse)
#     c) Run the code in ex41_prelim.R to import the data for this exercise
source("./exercise/ex41_prelim.R")
##
##
##
# 1. a. Take a look at d1, which comes from the ESTIMATES sheet of WPP2019_POP_F01_1
#    b. Reshape d1 to a tidy format with dimensions 20,235 x 4 columns (name, code,
d1

# 2. a. Take a look at d2, which comes from the ESTIMATES sheet of WPP2019_POP_F07_1
#    b. Reshape d2 to a tidy format with dimensions 79,065 x 5 columns (name, code,
d2

# 3. a. Take a look at d3, which comes from 2015_WPPDataset_AllVariables.xls
#    b. Reshape d3 to a tidy format with dimensions 4,577 x 5 columns (name, code, r
d3

# 4. a. Take a look at d4, which comes from the `Mid-2015 Persons` sheet of SAPE18DT
#    b. Reshape d4 to a tidy format with dimensions 143,431 x 5. columns (code, name
d4
```

Split and Combine

- The `separate()` and `unite()` functions help you split and combine cells to place a single, complete value in each cell.
- The `separate_rows()` function helps tidy a variable containing observations with multiple delimited values

separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", ...**)**

Separate each cell in a column to make several columns.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172
B	2000	80K	174
C	1999	212K	1T
C	2000	213K	1T

separate(table3, rate, into = c("cases", "pop"))

unite(data, col, ..., sep = "_", remove = TRUE**)**

Collapse cells across several columns to make a single column.

table5

country	century	year
Afghan	19	99
Afghan	20	0
Brazil	19	99
Brazil	20	0
China	19	99
China	20	0

→

country	year
Afghan	1999
Afghan	2000
Brazil	1999
Brazil	2000
China	1999
China	2000

unite(table5, century, year, col = "year", sep = "")

separate_rows(data, ..., sep = "[^[:alnum:].]+", convert = FALSE**)**

Separate each cell in a column to make several rows. Also **separate_rows_()**.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

country	year	rate
A	1999	0.7K
A	1999	19M
A	2000	2K
A	2000	20M
B	1999	37K
B	1999	172M
B	2000	80K
B	2000	174M
C	1999	212K
C	1999	1T
C	2000	213K
C	2000	1T

separate_rows(table3, rate)

Separate

- The `separate()` function splits a character column into multiple columns based on pattern.
- For example we can work create new columns for the start and end year in the tfr data by separating on the `"-"` character.
- Notice by default these turn into `character` vectors

```
> separate(df2, period, into = c("year0", "year5"), sep = "-")
# A tibble: 3,178 x 5
   name    code year0 year5   tfr
   <chr>  <dbl> <chr> <chr> <dbl>
 1 Africa   903 1950  1955   6.57
 2 Africa   903 1955  1960   6.62
 3 Africa   903 1960  1965   6.70
 4 Africa   903 1965  1970   6.71
 5 Africa   903 1970  1975   6.70
 6 Africa   903 1975  1980   6.64
 7 Africa   903 1980  1985   6.50
 8 Africa   903 1985  1990   6.19
 9 Africa   903 1990  1995   5.72
10 Africa   903 1995  2000   5.35
# ... with 3,168 more rows
```

Separate

- We can convert to numeric using the `convert` argument
- Keep the original column using `remove`
- The default for `sep` argument is `sep = "[^[:alnum:]]+"`. This searches regular expression for any character that is *not* alpha-numeric (a-z, A-Z, 0-9), i.e. *, −, , ., ,, _ and so on.
  - Hence do not need to state `sep =` explicitly for the period column

```
> separate(df2, period, into = c("year0", "year5"), convert = TRUE, remove = FALSE)
# A tibble: 3,178 x 6
   name    code period    year0 year5   tfr
   <chr>  <dbl> <chr>     <int> <int> <dbl>
 1 Africa   903 1950-1955  1950  1955  6.57
 2 Africa   903 1955-1960  1955  1960  6.62
 3 Africa   903 1960-1965  1960  1965  6.70
 4 Africa   903 1965-1970  1965  1970  6.71
 5 Africa   903 1970-1975  1970  1975  6.70
 6 Africa   903 1975-1980  1975  1980  6.64
 7 Africa   903 1980-1985  1980  1985  6.50
 8 Africa   903 1985-1990  1985  1990  6.19
 9 Africa   903 1990-1995  1990  1995  5.72
10 Africa   903 1995-2000  1995  2000  5.35
# ... with 3,168 more rows
```

Expand and Complete

- The `expand()` and `complete()` functions help you quickly create tables with combinations of values.

```
> # create data
> d0 <- tibble(year = c(2010, 2000, 2015), alpha3 = c("CHN", "KOR", "JPN"))
> d0
# A tibble: 3 x 2
   year alpha3
  <dbl> <chr>
1  2010 CHN
2  2000 KOR
3  2015 JPN
```

Expand and Complete

- Expand can save you lots of time and mistakes when you want all combinations.

```
> # expand for all combinations
> expand(d0, year, alpha3)
# A tibble: 9 x 2
   year alpha3
  <dbl> <chr>
1  2000 CHN
2  2000 JPN
3  2000 KOR
4  2010 CHN
5  2010 JPN
6  2010 KOR
7  2015 CHN
8  2015 JPN
9  2015 KOR
```

Expand and Complete

- The `full_seq()` function in `tidyr` helps complete data frames based on existing columns

```
> # complete
> complete(d0, alpha3, year = full_seq(x = year, period = 5))
# A tibble: 12 x 2
   alpha3  year
   <chr>  <dbl>
 1 CHN     2000
 2 CHN     2005
 3 CHN     2010
 4 CHN     2015
 5 JPN     2000
 6 JPN     2005
 7 JPN     2010
 8 JPN     2015
 9 KOR     2000
10 KOR     2005
11 KOR     2010
12 KOR     2015
```

Missing Values

- The `drop_na()` function filters out rows with one or more missing values
- The `fill()` function replaces missing values going up or down a named column.
- The `replace_na()` function replaces missing values given in a names `list`

drop_na(data, ...**)**

Drop rows containing NA's in … columns.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
D	3

drop_na(x, x2)

fill(data, ..., .direction = c("down", "up")**)**

Fill in NA's in … columns with most recent non-NA values.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	1
C	1
D	3
E	3

fill(x, x2)

replace_na(data, replace = list(), ...**)**

Replace NA's by column.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	2
C	2
D	3
E	2

replace_na(x,list(x2 = 2))

tidyr

Function	Description
`pivot_longer()`	Stack columns of same variables to a longer data set
`pivot_wider()`	Spread rows of different variables to wider data set.
`seperate()`	Separate one column into multiple columns.
`unite()`	Unite multiple columns into one.
`separate_rows()`	Separate a collapsed column into multiple rows
`fill()`	Fill in missing values.
`drop_na()`	Drop rows containing missing values
`replace_na()`	Replace missing values

Exercise 2 (ex42.R)

```
# 0.  a) Check your working directory is in the course folder. Load the .Rproj file

#     b) Load the tidyverse package

#     c) Run the code in ex42_prelim.R to import the data for this exercise

##
##
##
# 1.  Modify d2 to separate the age column so that
#     a) two new columns for the first and last age in each group are created (calle
#     b) the original age column is not removed
#     c) the new columns are integer values

d2
# 2.  Check out the last few rows in d3

# 3.  Modify d3 to drop the rows with missing values

# 4.  Uncomment the following code to save the immigration policy rationale data bas
# d7 <- d3 %>%
#   filter(issue == "Rationale for current immigration policy") %>%
#   select(name, policy)
# d7
# 5.  Modify d7 using the separate_rows() function to create new rows for each polic
```

Data Grammar

- Once your data is in a tidy format, more often than not we will want to make some transformations.
- The `dplyr` package is becoming increasingly popular for data manipulation.
  - Built by Hadley Wickham and Romain Francois
  - Fast, highly expressive, and open-minded about how your data is stored.
  - Originated out of `plyr`, which works for many different inputs (e.g., `arrays`, `data.frames`, `lists`)
  - `dplyr` is focused on data frames only.

Data Grammar

- There are six `dplyr` functions that you will use to do the vast majority of data manipulations:
  - `select()` to pick variables by their names
  - `filter()` to pick observations by their values
  - `arrange()` to reorder the rows
  - `mutate()` to create new variables with functions of existing variables
  - `summarise()` to collapse many values down to a single summary
- These can all be used in conjunction with `group_by()` (sixth function) which changes the scope of each function from operating on the entire data set to operating on it group-by-group.
- These six functions provide the verbs for a language of data manipulation.
- All verbs work similarly:
  - The first argument is a data frame.
  - The subsequent arguments describe what to do with the data frame.
  - You no longer have to name the columns in the data frame directly with $.
  - Will output a new data frame.

Extract Variables

- The `select()` function can be used to subset the variables or columns of a data set.

```
> # for example
> select(df2, name, period, tfr)
# A tibble: 3,178 x 3
   name    period      tfr
   <chr>   <chr>     <dbl>
 1 Africa  1950-1955  6.57
 2 Africa  1955-1960  6.62
 3 Africa  1960-1965  6.70
 4 Africa  1965-1970  6.71
 5 Africa  1970-1975  6.70
 6 Africa  1975-1980  6.64
 7 Africa  1980-1985  6.50
 8 Africa  1985-1990  6.19
 9 Africa  1990-1995  5.72
10 Africa  1995-2000  5.35
# ... with 3,168 more rows
```

Extract Variables

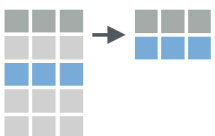
- As demonstrated with `pivot_longer()` we can use standard indexing methods in R:
  - `:` for sequence
  - `-` to drop columns
- These also work with names and column numbers
- There are helpers functions to aid selections, e.g.
  - `contains(match)`
  - `starts_with(match)`
  - `matches(regular_expression)`

Extract Variables

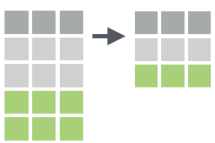
```
> # these are all the same...
> # select(df2, name, period:tfr)
> # select(df2, 1, 3, 4)
> # select(df2, -2)
> # select(df2, -starts_with("c"))
> # select(df2, -contains("c"))
> select(df2, -code)
# A tibble: 3,178 x 3
   name   period      tfr
   <chr>  <chr>     <dbl>
 1 Africa 1950-1955  6.57
 2 Africa 1955-1960  6.62
 3 Africa 1960-1965  6.70
 4 Africa 1965-1970  6.71
 5 Africa 1970-1975  6.70
 6 Africa 1975-1980  6.64
 7 Africa 1980-1985  6.50
 8 Africa 1985-1990  6.19
 9 Africa 1990-1995  5.72
10 Africa 1995-2000  5.35
# ... with 3,168 more rows
```

Extract Cases

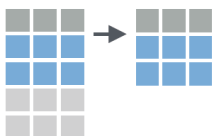
- The `filter()` function returns rows with matching conditions.
- The `distinct()` function returns only unique/distinct rows
- The `slice()` function returns the observations requested.
- The `top_n()` returns the first `n` observations. Similar to `head()`



filter(.data, …**)**
Extract rows that meet logical criteria.



distinct(.data, ..., .keep_all = FALSE**)**
Remove rows with duplicate values.



slice(.data, …**)**
Select rows by position.

top_n(*x, n, wt***)**
Select and order top n entries (by group if grouped data).

Extract Cases

- The `filter()` function in `dplyr` is a popular tool for creating subsets.
- Takes logical expressions and returns the rows for which all are `TRUE`

```
> filter(df2, period == "2010-2015", name == "China")
# A tibble: 1 x 4
  name    code period      tfr
  <chr>  <dbl> <chr>     <dbl>
1 China    156 2010-2015  1.64
```

Extract Cases

- The %in% is useful for matching multiple cases in the same variable.

```
> # can also use `<` `>` `>=` `<=` `!=`
> filter(df2, name == "China", tfr < 2)
# A tibble: 6 x 4
  name    code period      tfr
  <chr>  <dbl> <chr>     <dbl>
1 China    156 1990-1995  1.83
2 China    156 1995-2000  1.62
3 China    156 2000-2005  1.61
4 China    156 2005-2010  1.62
5 China    156 2010-2015  1.64
6 China    156 2015-2020  1.69
>
> # multiple matches on the same variable we can also use %in%
> filter(df2, period == "2010-2015", code %in% c(156, 410))
# A tibble: 2 x 4
  name                code period      tfr
  <chr>              <dbl> <chr>     <dbl>
1 China                156 2010-2015  1.64
2 Republic of Korea    410 2010-2015  1.23
```

Pipelines

- The pipeline operator (%>%) from the `magrittr` package by Stefan Bache is part of the tidyverse and has two big benefits
  - You can tie together R commands and thus avoid nesting code.
  - The syntax leads to code that is much easier to write and explain.
- The pipe operator takes whatever is before the pipe and drops it in as the **first** argument of the next function call.
  - e.g. `x %>% mean()` is the same as `mean(x)`
- Can keep using pipe operator to chain together code
  - e.g. `x %>% log() %>% sqrt() %>% mean()`

Pipelines

- Pipe operators allow you to easily build up readable code.
- It is good practice to start a new line after each pipe. RStudio will automatically indent
- When you see the pipe operator, %>%, read it as 'then'.
- You can still specify other (second, third, etc.) arguments (i.e. any but the first) in each function.

```
> # for example
> df2 %>%
+   filter(name == "China", tfr < 2) %>%
+   tail(n = 5)
# A tibble: 5 x 4
  name    code period      tfr
  <chr>  <dbl> <chr>     <dbl>
1 China    156 1995-2000  1.62
2 China    156 2000-2005  1.61
3 China    156 2005-2010  1.62
4 China    156 2010-2015  1.64
5 China    156 2015-2020  1.69
> # same as
> # tail(filter(data = df2, name == "China"), n = 5)
```

Pipelines

- The `.` is called a placeholder. It represents the results from everything higher up the code chain. `d %>% f(y, z = .)` is equivalent to `f(y, z = d)`, e.g. `d %>% lm(formula = a ~ b, data = .)` is the same as `lm(formula = a ~ b, data = d)`
- It is useful when you want pass the results from everything higher up the code chain to the an argument that is **not** by default the **first** in the function.
  - Not the case for almost all `tidyverse` functions

Pipelines

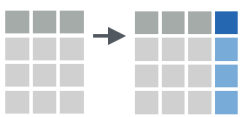
```
> df2 %>%
+   filter(name == "China") %>%
+   lm(formula = tfr ~ lag(tfr), data = .)

Call:
lm(formula = tfr ~ lag(tfr), data = .)

Coefficients:
(Intercept)     lag(tfr)
    0.07935      0.88011
>
> df2 %>%
+   filter(name == "China") %>%
+   lm(formula = tfr ~ lag(tfr), data = .) %>%
+   residuals() %>%
+   sd()
[1] 0.6666045
```

Exercise 3 (ex43.R)

```
# 0.  a) Check your working directory is in the course folder. Load the .Rproj file

#     b) Load the tidyverse package

#     c) Run the code in ex43_prelim.R to import the data for this exercise

##
##
##
# 1.  Display the observations in d1 from Austria

# 2.  Display the distinct observations in d1 of each area name and code (285 rows)

# 3.  Display the observations in d1 using pipes (%>%)
#     a) from Austria
#     b) the name, year and pop variables only



# 4.  Display the observations in d2 using pipes
#     a) from Philippines in 2015
#     b) the name, age and pop variables only
```

Data Creation

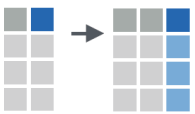
- The `mutate()` function for adding new or replacing existing columns.
- The `transmutate()` function for dropping other columns
- The `rename()` function for changing column names
- The `mutate_all()` modifies all columns
- The `mutate_if()` modifies all columns selected



mutate(.data, …**)**
Compute new column(s).

transmute(.data, …**)**
Compute new column(s), drop others.

rename(.data, …**)**
Rename columns.



mutate_at(.tbl, .cols, .funs, ...**)**
Apply funs to specific columns. Use with **funs()**, **vars()** and the helper functions for select().

mutate_if(.tbl, .predicate, .funs, ...**)**
Apply funs to all columns of one type. Use with **funs().**

mutate_all(.tbl, .funs, ...**)**
Apply funs to every column. Use with **funs()**.

Adding New Variables

- The `mutate()` function is used when you want to create a new column.
  - Can overwrite new columns if you want to replace their values.
  - Can create new columns, for example to code regions or countries (where country code greater than or equal to 900)

```
> df2 %>%
+   mutate(tfr = round(tfr,1),
+          location_type = ifelse(test = code >= 900, yes = "region", no = "nation"
# A tibble: 3,178 x 5
   name     code period      tfr location_type
   <chr>   <dbl> <chr>     <dbl> <chr>
 1 Africa    903 1950-1955   6.6 region
 2 Africa    903 1955-1960   6.6 region
 3 Africa    903 1960-1965   6.7 region
 4 Africa    903 1965-1970   6.7 region
 5 Africa    903 1970-1975   6.7 region
 6 Africa    903 1975-1980   6.6 region
 7 Africa    903 1980-1985   6.5 region
 8 Africa    903 1985-1990   6.2 region
 9 Africa    903 1990-1995   5.7 region
10 Africa    903 1995-2000   5.4 region
# ... with 3,168 more rows
```

Adding New Variables

- The `case_when()` function is used when you want to create a new column based on multiple `ifelse()` statements
- Uses a sequence of two-sided formulas
  - The left hand side (LHS) determines which values match this case. Must be a logical vector
  - The right hand side (RHS) provides the replacement value.

```
> df2 %>%
+   mutate(location_type = case_when(
+     code < 900 ~ "country",
+     code %in% code[1:6] ~ "area",
+     code > 900 ~ "region")
+   )
# A tibble: 3,178 x 5
   name    code period      tfr location_type
   <chr>  <dbl> <chr>     <dbl> <chr>
 1 Africa   903 1950-1955  6.57 area
 2 Africa   903 1955-1960  6.62 area
 3 Africa   903 1960-1965  6.70 area
 4 Africa   903 1965-1970  6.71 area
 5 Africa   903 1970-1975  6.70 area
 6 Africa   903 1975-1980  6.64 area
 7 Africa   903 1980-1985  6.50 area
 8 Africa   903 1985-1990  6.19 area
```

Recoding Variables

- The `recode` or `recode_factor()` function takes
  - `.x` a column of characters or factors
  - `...` a set of replacements in the style `"old label" = "new label"`

```
> df2 %>%
+   mutate(name = recode(name,
+                        "Africa" = "Continent: Africa",
+                        "Eastern Africa" = "Region: Eastern Africa"))
# A tibble: 3,178 x 4
   name                 code period      tfr
   <chr>               <dbl> <chr>     <dbl>
 1 Continent: Africa     903 1950-1955  6.57
 2 Continent: Africa     903 1955-1960  6.62
 3 Continent: Africa     903 1960-1965  6.70
 4 Continent: Africa     903 1965-1970  6.71
 5 Continent: Africa     903 1970-1975  6.70
 6 Continent: Africa     903 1975-1980  6.64
 7 Continent: Africa     903 1980-1985  6.50
 8 Continent: Africa     903 1985-1990  6.19
 9 Continent: Africa     903 1990-1995  5.72
10 Continent: Africa     903 1995-2000  5.35
# ... with 3,168 more rows
```

Recoding Variables

- The `cut()` function divides the range of `x` into intervals and codes the values in x according to which interval they fall.

```
> df2 %>%
+   filter(period == last(period),
+          code < 900) %>%
+   select(-code, -period) %>%
+   mutate(tfr_interval4 = cut(x = tfr, breaks = c(min(tfr), 2, 4, 6, max(tfr))),
+          tfr_interval7 = cut(x = tfr, breaks = seq(from = 0, to = 7, by = 1)))
# A tibble: 201 x 4
   name          tfr tfr_interval4 tfr_interval7
   <chr>       <dbl> <fct>         <fct>
 1 Burundi      5.45 (4,6]         (5,6]
 2 Comoros      4.24 (4,6]         (4,5]
 3 Djibouti     2.76 (2,4]         (2,3]
 4 Eritrea      4.1  (4,6]         (4,5]
 5 Ethiopia     4.3  (4,6]         (4,5]
 6 Kenya        3.52 (2,4]         (3,4]
 7 Madagascar   4.11 (4,6]         (4,5]
 8 Malawi       4.25 (4,6]         (4,5]
 9 Mauritius    1.39 (1.11,2]      (1,2]
10 Mayotte      3.73 (2,4]         (3,4]
# ... with 191 more rows
```

Rename

- If you want to rename the column, rather than create a new column and delete the old one there is a `rename()` function
- Works in a opposite order than `recode()`, i.e. `new_name = old_name`.
- Does not require " " for names
- Can use column index i.e. `new_name = column number`.

```
> df2 %>%
+   rename(fertility_rate = tfr) %>%
+   head(n = 2)
# A tibble: 2 x 4
  name    code period    fertility_rate
  <chr>  <dbl> <chr>              <dbl>
1 Africa   903 1950-1955           6.57
2 Africa   903 1955-1960           6.62
> df2 %>%
+   rename(country_code = 2) %>%
+   head(n = 2)
# A tibble: 2 x 4
  name   country_code period      tfr
  <chr>         <dbl> <chr>     <dbl>
1 Africa          903 1950-1955  6.57
2 Africa          903 1955-1960  6.62
```

Data Creation

```
> df2 %>%
+   filter(name == "China", tfr < 2) %>%
+   mutate_if(is.numeric, ~ (.*100))
# A tibble: 6 x 4
  name    code period      tfr
  <chr>  <dbl> <chr>     <dbl>
1 China 15600 1990-1995   183
2 China 15600 1995-2000   162
3 China 15600 2000-2005   161
4 China 15600 2005-2010   162
5 China 15600 2010-2015   164
6 China 15600 2015-2020   169
>
> df2 %>%
+   filter(name == "China", tfr < 2) %>%
+   select(code, tfr) %>%
+   mutate_all(~ (.*100))
# A tibble: 6 x 2
   code   tfr
  <dbl> <dbl>
1 15600   183
2 15600   162
3 15600   161
4 15600   162
5 15600   164
```

Arranging Data

- The arrange function sorts data frames according the value of the cells.

```
> df2 %>%
+   filter(name == "China") %>%
+   arrange(tfr)
# A tibble: 14 x 4
   name    code period      tfr
   <chr> <dbl> <chr>     <dbl>
 1 China   156 2000-2005  1.61
 2 China   156 1995-2000  1.62
 3 China   156 2005-2010  1.62
 4 China   156 2010-2015  1.64
 5 China   156 2015-2020  1.69
 6 China   156 1990-1995  1.83
 7 China   156 1980-1985  2.52
 8 China   156 1985-1990  2.73
 9 China   156 1975-1980  3.01
10 China   156 1970-1975  4.85
11 China   156 1955-1960  5.48
12 China   156 1950-1955  6.11
13 China   156 1960-1965  6.15
14 China   156 1965-1970  6.3
```

Arranging Data

- Can arrange in descending order using `desc`

```
> df2 %>%
+   filter(name == "China") %>%
+   arrange(desc(tfr))
# A tibble: 14 x 4
   name    code period      tfr
   <chr> <dbl> <chr>     <dbl>
 1 China   156 1965-1970  6.3
 2 China   156 1960-1965  6.15
 3 China   156 1950-1955  6.11
 4 China   156 1955-1960  5.48
 5 China   156 1970-1975  4.85
 6 China   156 1975-1980  3.01
 7 China   156 1985-1990  2.73
 8 China   156 1980-1985  2.52
 9 China   156 1990-1995  1.83
10 China   156 2015-2020  1.69
11 China   156 2010-2015  1.64
12 China   156 1995-2000  1.62
13 China   156 2005-2010  1.62
14 China   156 2000-2005  1.61
```

Arranging Data

- If the column being sorted is a `character` will use alphabetical order.
- This is where `factor` come in useful.
- Consider the UN population data by age group...

```
> df5 <- read_excel("../data/WPP2019_POP_F07_1_POPULATION_BY_AGE_BOTH_SEXES.xlsx",
+                   skip = 16, na="\U2026") %>%
+   select(3, 5, 8, contains("-"), contains("+")) %>%
+   rename(name = 1,
+          code = 2,
+          year = 3) %>%
+   pivot_longer(cols = -("name":"year"), names_to = "age_grp", values_to = "pop")
```

Arranging Data

```
> df5
# A tibble: 80,325 x 5
   name    code  year age_grp     pop
   <chr> <dbl> <dbl> <chr>     <dbl>
 1 WORLD   900  1950 0-4     338497.
 2 WORLD   900  1950 5-9     270084.
 3 WORLD   900  1950 10-14   261028.
 4 WORLD   900  1950 15-19   239488.
 5 WORLD   900  1950 20-24   222896.
 6 WORLD   900  1950 25-29   196163.
 7 WORLD   900  1950 30-34   169236.
 8 WORLD   900  1950 35-39   164473.
 9 WORLD   900  1950 40-44   147563.
10 WORLD   900  1950 45-49   128450.
# ... with 80,315 more rows
```

Arranging Data

- Consider sorting the China data by age
- Gives an unfriendly order

```
> df5 %>%
+   filter(name == "China", year == 2015) %>%
+   arrange(age_grp)
# A tibble: 21 x 5
   name    code  year age_grp      pop
   <chr> <dbl> <dbl> <chr>      <dbl>
 1 China   156  2015 0-4       86938.
 2 China   156  2015 10-14     82638.
 3 China   156  2015 100+         42.0
 4 China   156  2015 15-19     87675.
 5 China   156  2015 20-24     98594.
 6 China   156  2015 25-29    129455.
 7 China   156  2015 30-34    100711.
 8 China   156  2015 35-39     96973.
 9 China   156  2015 40-44    120968.
10 China   156  2015 45-49    125209.
# ... with 11 more rows
```

Arranging Data

- We can use mutate to change the age_grp column to a factor as:

```
> df5 <- mutate(df5, age_grp = fct_inorder(age_grp))
>
> # now arrange works well...
> df5 %>%
+   filter(name == "China", year == 2015) %>%
+   arrange(age_grp)
# A tibble: 21 x 5
   name    code  year age_grp     pop
   <chr> <dbl> <dbl> <fct>     <dbl>
 1 China   156  2015 0-4      86938.
 2 China   156  2015 5-9      84413.
 3 China   156  2015 10-14    82638.
 4 China   156  2015 15-19    87675.
 5 China   156  2015 20-24    98594.
 6 China   156  2015 25-29   129455.
 7 China   156  2015 30-34   100711.
 8 China   156  2015 35-39    96973.
 9 China   156  2015 40-44   120968.
10 China   156  2015 45-49   125209.
# ... with 11 more rows
```

Exercise 4 (ex44.R)

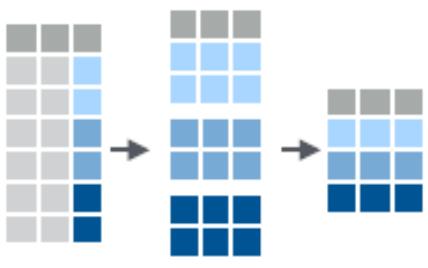
```
# 0.  a) Check your working directory is in the course folder. Load the .Rproj file

#     b) Load the tidyverse packages

#     c) Run the code in ex44_prelim.R to import the data for this exercise

##
##
##
# 1. Modify d2 to
#    a) add a `location_type` variable with value "region" if region and "nation" if
#    (Hint: use ifelse() function, all countries have codes less than 900)
#    b) change population from thousands to integer (Hint: * 1000)
#    c) change age to factor with levels in age order (Hint: use the fct_inorder() f
```

d2

```
# 2. Display the observations in d2 using pipes
#    a) from nations, in 2020, and age group 100+
#    b) with the pop column renamed to pop_centurion
#    c) sorted in descending order based on pop_centurion variable
#    d) the name and pop_centurion column only
```

Grouped

- Group data adds some extra information to the tibble which is helpful when we want to do some extra calculations.
- Changes unit of analysis from the complete data set to groups.
- We can use it with
  - `summarise()` to create a new data frame of summarizing each group.
  - `mutate()` to create new columns to create summary variables



group_by(.data, ..., add = FALSE)
Returns copy of table grouped by …

ungroup(x, ...)
Returns ungrouped copy of table.

Grouped Summary

- The `summarise()` (or `summarize()`) function collapses a data frame to a single row:
- Paired with `group_by()` it becomes very powerful.

summarise() applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

Counts

dplyr::**n()** - number of values/rows
dplyr::**n_distinct()** - # of uniques
sum(!is.na()) - # of non-NA's

Location

mean() - mean, also **mean(!is.na())**
median() - median

Logicals

mean() - Proportion of TRUE's
sum() - # of TRUE's

Position/Order

dplyr::**first()** - first value
dplyr::**last()** - last value
dplyr::**nth()** - value in nth location of vector

Grouped Summary

```
> # original data
> df5
# A tibble: 80,325 x 5
   name    code  year age_grp     pop
   <chr> <dbl> <dbl> <fct>     <dbl>
 1 WORLD   900  1950 0-4     338497.
 2 WORLD   900  1950 5-9     270084.
 3 WORLD   900  1950 10-14   261028.
 4 WORLD   900  1950 15-19   239488.
 5 WORLD   900  1950 20-24   222896.
 6 WORLD   900  1950 25-29   196163.
 7 WORLD   900  1950 30-34   169236.
 8 WORLD   900  1950 35-39   164473.
 9 WORLD   900  1950 40-44   147563.
10 WORLD   900  1950 45-49   128450.
# ... with 80,315 more rows
```

Grouped Summary

```
> # summarise over all age groups for China and Japan
> df5 %>%
+   filter(year == 2015, name %in% c("China", "Japan"))  %>%
+   summarise(pop_sum = sum(pop, na.rm = TRUE))
# A tibble: 1 x 1
   pop_sum
     <dbl>
1 1534833.
>
> # summarise over all age groups for each of China and Japan
> df5 %>%
+   filter(year == 2015, name %in% c("China", "Japan"))  %>%
+   drop_na() %>%
+   group_by(name) %>%
+   summarise(pop_sum = sum(pop),
+             n = n(),
+             under5 = first(pop))
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 2 x 4
  name   pop_sum     n under5
  <chr>    <dbl> <int>  <dbl>
1 China 1406848.    21 86938.
2 Japan  127985.    21  5395.
```

Multiple Groups

- Produce group summaries on multiple sub groups.

```
> df5 %>%
+   filter(code < 900) %>%
+   group_by(name, year) %>%
+   summarise(pop_sum = sum(pop, na.rm = TRUE))
`summarise()` regrouping output by 'name' (override with `.groups` argument)
# A tibble: 3,015 x 3
# Groups:   name [201]
   name          year pop_sum
   <chr>        <dbl>   <dbl>
 1 Afghanistan  1950    7752.
 2 Afghanistan  1955    8271.
 3 Afghanistan  1960    8997.
 4 Afghanistan  1965    9956.
 5 Afghanistan  1970   11174.
 6 Afghanistan  1975   12689.
 7 Afghanistan  1980   13357.
 8 Afghanistan  1985   11938.
 9 Afghanistan  1990   12412.
10 Afghanistan  1995   18111.
# ... with 3,005 more rows
```

Grouped Mutate

```
> # original data
> df2
# A tibble: 3,178 x 4
   name    code period      tfr
   <chr>  <dbl> <chr>     <dbl>
 1 Africa   903 1950-1955  6.57
 2 Africa   903 1955-1960  6.62
 3 Africa   903 1960-1965  6.70
 4 Africa   903 1965-1970  6.71
 5 Africa   903 1970-1975  6.70
 6 Africa   903 1975-1980  6.64
 7 Africa   903 1980-1985  6.50
 8 Africa   903 1985-1990  6.19
 9 Africa   903 1990-1995  5.72
10 Africa   903 1995-2000  5.35
# ... with 3,168 more rows
```

Grouped Mutate

```
> # add columns for the all time mean tfr, max tfr and lagged tfr
> df2 %>%
+   filter(name %in% c("China", "Japan")) %>%
+   group_by(name) %>%
+   mutate(mean_tfr = mean(tfr),
+          max_tfr = max(tfr),
+          lag_tfr = lag(tfr))
# A tibble: 28 x 7
# Groups:   name [2]
   name    code period      tfr mean_tfr max_tfr lag_tfr
   <chr> <dbl> <chr>     <dbl>    <dbl>   <dbl>   <dbl>
 1 China   156 1950-1955  6.11     3.37     6.3   NA
 2 China   156 1955-1960  5.48     3.37     6.3    6.11
 3 China   156 1960-1965  6.15     3.37     6.3    5.48
 4 China   156 1965-1970  6.3      3.37     6.3    6.15
 5 China   156 1970-1975  4.85     3.37     6.3    6.3
 6 China   156 1975-1980  3.01     3.37     6.3    4.85
 7 China   156 1980-1985  2.52     3.37     6.3    3.01
 8 China   156 1985-1990  2.73     3.37     6.3    2.52
 9 China   156 1990-1995  1.83     3.37     6.3    2.73
10 China   156 1995-2000  1.62     3.37     6.3    1.83
# ... with 18 more rows
```

Exercise 5 (ex45.R)

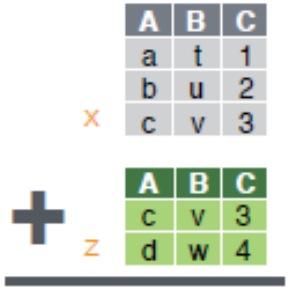
```
# 0.  a) Check your working directory is in the course folder. Load the .Rproj file

#     b) Load the tidyverse packages

#     c) Run the code in ex45_prelim.R to import the data for this exercise

##
##
##
# 1. Display the observations in d1 using pipes
#    a) from Philippines, Indonesia and Malaysia
#    b) grouped by name
#    c) with a new pop_lag variable for the population in the year previous in each
#    d) with a new pop_rate variable based on (pop - pop_lag)/pop




# 2. Display the observations in d3 using pipes
#    a) grouped by region
#    b) with a new variable called n_nations summarising the number of countries in
#       (Hint: use n_distinct(name))
```

Relational Data

- Data analysis rarely involves only a single table of data.
- Typically you want to combine data to answer the questions that you are interested in.
- Data joins can involve
  - Joining data with different variables (columns)
  - Joining data with different observations (rows)

Combine Cases

- The `bind_rows()` will fill `NA` values for columns that are missing from one of the data sets.
- The `intersect()`, `union()` and `setdiff()` functions look at matching rules for observations (rows).



Use **bind_rows()** to paste tables below each other as they are.

DF	A	B	C
x	a	t	1
x	b	u	2
x	c	v	3
z	c	v	3
z	d	w	4

bind_rows(..., .id = NULL)
Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured)

A	B	C
c	v	3

intersect(x, y, ...)
Rows that appear in both x and z.

A	B	C
a	t	1
b	u	2

setdiff(x, y, ...)
Rows that appear in x but not z.

A	B	C
a	t	1
b	u	2
c	v	3
d	w	4

union(x, y, ...)
Rows that appear in x or z. (Duplicates removed). **union_all()** retains duplicates.

Combine Cases

```
> d1
# A tibble: 3 x 3
  A     B         C
  <chr> <chr> <int>
1 a     t         1
2 b     u         2
3 c     v         3
> d2
# A tibble: 3 x 3
  A     B         D
  <chr> <chr> <int>
1 a     t         3
2 b     u         2
3 d     w         1
> bind_rows(d1, d2)
# A tibble: 6 x 4
  A     B         C     D
  <chr> <chr> <int> <int>
1 a     t         1    NA
2 b     u         2    NA
3 c     v         3    NA
4 a     t        NA     3
5 b     u        NA     2
6 d     w        NA     1
```

Combine Cases

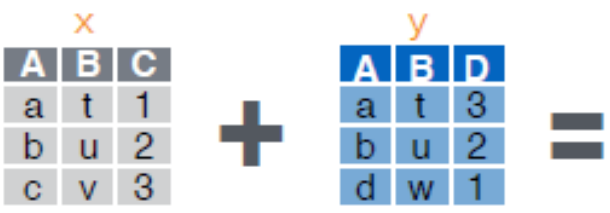
```
> d1
# A tibble: 3 x 3
  A     B         C
  <chr> <chr> <int>
1 a     t         1
2 b     u         2
3 c     v         3
> d3
# A tibble: 2 x 3
  A     B         C
  <chr> <chr> <int>
1 c     v         3
2 d     w         4
> bind_rows(d1, d3)
# A tibble: 5 x 3
  A     B         C
  <chr> <chr> <int>
1 a     t         1
2 b     u         2
3 c     v         3
4 c     v         3
5 d     w         4
```

Exercise 6 (ex46.R)

```
# 0.  a) Check your working directory is in the course folder. Load the .Rproj file
getwd()
#     b) Load the tidyverse and readxl packages

##
##
##
# 1. Using WPP2019_POP_F07_1_POPULATION_BY_AGE_BOTH_SEXES on future population sizes
#    a) the 1st sheet and save as r1
#    b) the 2st sheet and save as r2
#    c) print r1
#    d) print r2
r1 <- #####(path = "./data/WPP2019_POP_F07_1_POPULATION_BY_AGE_BOTH_SEXES.xlsx",
                  sheet = 1, skip = 16, na = "\U2026")
##### <- read_excel(path = "#####",
                  sheet = #####, skip = 16, na = "\U2026")
r1
r2
# 2. Bind the rows of r1 and r2 and call the result r3

# 3. a) Q: How many rows in r1
#       A:
#    b) Q: How many rows in r2
#       A:
#    b) Q: How many rows in r3
```

Combine Variables

- The `bind_cols()` function is the simplest way to combine variables
  - The order of observations (rows) in both data MUST BE THE SAME
  - Can be rather dangerous, better to use joins (next)



Use **bind_cols()** to paste tables beside each other as they are.

A	B	C	A	B	D
a	t	1	a	t	3
b	u	2	b	u	2
c	v	3	d	w	1

bind_cols(...)

Returns tables placed side by side as a single table.

BE SURE THAT ROWS ALIGN.

Combine Variables

- When combining variables we need to know:
  - Observations in which variables match up (link the data sets together)
  - These are called "key" variables, similar to what we discussed when tidying the data.

x

A	B	C
a	t	1
b	u	2
c	v	3

+

y

A	B	D
a	t	3
b	u	2
d	w	1

=

A	B	C	D
a	t	1	3
b	u	2	2
c	v	3	NA

left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...)

Join matching values from y to x.

A	B.x	C	B.y	D
a	t	1	t	3
b	u	2	u	2
c	v	3	NA	NA

Use **by = c("col1", "col2")** to specify the column(s) to match on.

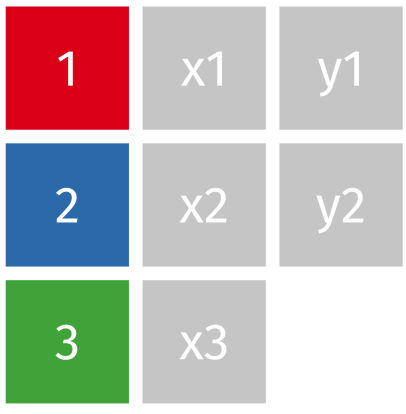
left_join(x, y, by = "A")

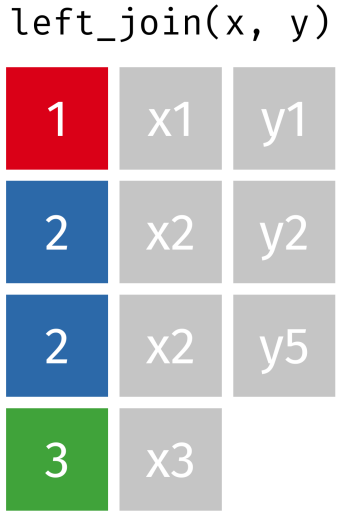
A.x	B.x	C	A.y	B.y
a	t	1	d	w
b	u	2	b	u
c	v	3	a	t

Use a named vector, **by = c("col1" = "col2")**, to match on columns with different names in each data set.

left_join(x, y, by = c("C" = "D"))

Combine Variables





source: https://github.com/gadenbuie/tidyexplain

Combine Variables

```
> d1
# A tibble: 3 x 3
  A     B         C
  <chr> <chr> <int>
1 a     t         1
2 b     u         2
3 c     v         3
> d2
# A tibble: 3 x 3
  A     B         D
  <chr> <chr> <int>
1 a     t         3
2 b     u         2
3 d     w         1
> # by default joins by common names
> left_join(d1, d2)
Joining, by = c("A", "B")
# A tibble: 3 x 4
  A     B         C     D
  <chr> <chr> <int> <int>
1 a     t         1     3
2 b     u         2     2
3 c     v         3    NA
```

Combine Variables

```
> d1
# A tibble: 3 x 3
  A     B         C
  <chr> <chr> <int>
1 a     t         1
2 b     u         2
3 c     v         3
> d2
# A tibble: 3 x 3
  A     B         D
  <chr> <chr> <int>
1 a     t         3
2 b     u         2
3 d     w         1
> # we can make this explicit
> left_join(d1, d2, by = c("A" = "A", "B" = "B"))
# A tibble: 3 x 4
  A     B         C     D
  <chr> <chr> <int> <int>
1 a     t         1     3
2 b     u         2     2
3 c     v         3    NA
```

Combine Variables

```
> d1
# A tibble: 3 x 3
  A     B         C
  <chr> <chr> <int>
1 a     t         1
2 b     u         2
3 c     v         3
> d4
# A tibble: 3 x 3
  AA    BB        D
  <chr> <chr> <int>
1 a     t         3
2 b     u         2
3 d     w         1
> # can join when names are not the same using the `by` argument
> left_join(d1, d4, by = c("A" = "AA", "B" = "BB"))
# A tibble: 3 x 4
  A     B         C     D
  <chr> <chr> <int> <int>
1 a     t         1     3
2 b     u         2     2
3 c     v         3    NA
```

Combine Variables

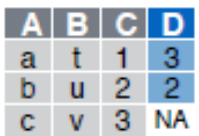
- When one table has duplicate observations in the key variable of `x` or `y`
  - Multiple matches will increase the number of rows
- When using `left_join()` this might not always be your intention.
  - Need to be careful with big data sets and check the number of rows `nrow()` of the new data set after the join and the original `x`.

Combine Variables

```
> d1
# A tibble: 3 x 3
  A     B         C
  <chr> <chr> <int>
1 a     t         1
2 b     u         2
3 c     v         3
> d5
# A tibble: 4 x 2
  A         D
  <chr> <int>
1 a         4
2 c         3
3 c         2
4 c         1
> # using %>% for joins
> d1 %>%
+   left_join(d5, by = "A")
# A tibble: 5 x 4
  A     B         C     D
  <chr> <chr> <int> <int>
1 a     t         1     4
2 b     u         2    NA
3 c     v         3     3
4 c     v         3     2
```

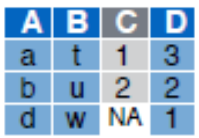
Combine Variables

- Usually `left_join()` function does the job
  - `right_join()` function switches the order of `x` and `y`. Mainly used with pipes.
  - `inner_join()` function is useful if you want to remove `NA` at the same time.
  - `full_join()` function is useful if you want to consider all possible combinations



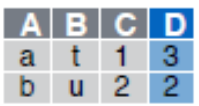
left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...)

Join matching values from y to x.



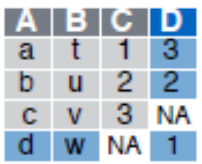
right_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),...)

Join matching values from x to y.



inner_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),...)

Join data. Retain only rows with matches.



full_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...)

Join data. Retain all values, all rows.

Exercise 7 (ex47.R)

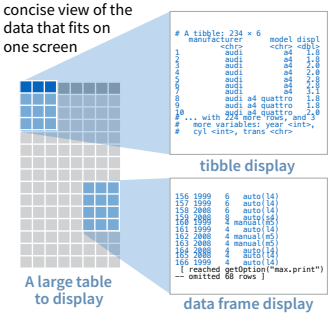
```
# 0.  a) Check your working directory is in the course folder. Load the .Rproj file

#     b) Load the tidyverse packages

#     c) Run the code in ex47_prelim.R to import the data for this exercise

##
##
##
# 1. View j1, j2, j3 and j4 loaded in from the prelim script
j1; j2; j3; j4
# 2. Create a new data set d0 from j1 that joins j1 with j2, j3 and j4 using pipes




d0
# 3. View j5

# 4. Join j5 onto d0

d0
# 5. View j6

# 6. Join j6 onto d0
```

RStudio Cheatsheets

Tibbles - an enhanced data frame

The **tibble** package provides a new S3 class for storing tabular data, the tibble. Tibbles inherit the data frame class, but improve three behaviors:

- **Subsetting** - [always returns a new tibble, [[and $ always return a vector.
- **No partial matching** - You must use full column names when subsetting
- **Display** - When you print a tibble, R provides a concise view of the data that fits on one screen



- Control the default appearance with options:

  options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf**)**

- View full data set with **View()** or **glimpse()**
- Revert to data frame with **as.data.frame()**

CONSTRUCT A TIBBLE IN TWO WAYS

tibble(…)
Construct by columns.
tibble(x = 1:3, y = c("a", "b", "c"))

tribble(…)
Construct by rows.
tribble(~x, ~y, 1, "a", 2, "b", 3, "c")

Both make this tibble

```
A tibble: 3 × 2
      x     y
  <int> <dbl>
1     1     a
2     2     b
3     3     c
```

as_tibble(x, …) Convert data frame to tibble.

enframe(x, name = "name", value = "value")
Convert named vector to a tibble

is_tibble(x) Test whether x is a tibble.

Tidy Data with Tidyr

Tidy data is a way to organize tabular data. It provides a consistent data structure across packages.

A table is tidy if:

Each **variable** is in its own **column**

&

Each **observation**, or **case**, is in its own **row**

Tidy data:

Makes variables easy to access as vectors

A * B -> C

Preserves cases during vectorized operations

Reshape Data - change the layout of values in a table

Use **gather()** and **spread()** to reorganize the values of a table into a new layout.

gather(data, key, value, ..., na.rm = FALSE, convert = FALSE, factor_key = FALSE**)**

Gather moves column names into a **key** column, gathering the column values into a single **value** column.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

key value

gather(table4a, `1999`, `2000`, key = "year", value = "cases")

spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL**)**

Spread moves the unique values of a **key** column into the column names, spreading the values of a **value** column across the new columns.

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

key value

spread(table2, type, count)

Handle Missing Values

drop_na(data, ...)
Drop rows containing NA's in … columns.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

x1	x2
A	1
D	3

drop_na(x, x2)

fill(data, ..., .direction = c("down", "up"))
Fill in NA's in … columns with most recent non-NA values.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

x1	x2
A	1
B	1
C	1
D	3
E	3

fill(x, x2)

replace_na(data, replace = list(), ...)
Replace NA's by column.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

x1	x2
A	1
B	2
C	2
D	3
E	2

replace_na(x, list(x2 = 2))

Expand Tables - quickly create tables with combinations of values

complete(data, ..., fill = list())
Adds to the data missing combinations of the values of the variables listed in …
complete(mtcars, cyl, gear, carb)

expand(data, ...)
Create new tibble with all possible combinations of the values of the variables listed in …
expand(mtcars, cyl, gear, carb)

Split Cells

Use these functions to split or combine cells into individual, isolated values.

separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", ...**)**

Separate each cell in a column to make several columns.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172
B	2000	80K	174
C	1999	212K	1T
C	2000	213K	1T

separate(table3, rate, into = c("cases", "pop"))

separate_rows(data, ..., sep = "[^[:alnum:].]+", convert = FALSE**)**

Separate each cell in a column to make several rows. Also **separate_rows_()**.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

country	year	rate
A	1999	0.7K
A	1999	19M
A	2000	2K
A	2000	20M
B	1999	37K
B	1999	172M
B	2000	80K
B	2000	174M
C	1999	212K
C	1999	1T
C	2000	213K
C	2000	1T

separate_rows(table3, rate)

unite(data, col, ..., sep = "_", remove = TRUE)

Collapse cells across several columns to make a single column.

table5

country	century	year
Afghan	19	99
Afghan	20	0
Brazil	19	99
Brazil	20	0
China	19	99
China	20	0

country	year
Afghan	1999
Afghan	2000
Brazil	1999
Brazil	2000
China	1999
China	2000

unite(table5, century, year, col = "year", sep = "")

RStudio Cheatsheets

Data Transformation with dplyr : : CHEAT SHEET

dplyr functions work with pipes and expect **tidy data**. In tidy data:

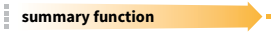
Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own **row**

pipes

x %>% f(y) becomes **f(x, y)**

Summarise Cases

These apply **summary functions** to columns to create a new table. Summary functions take vectors as input and return one value (see back).



summary function

summarise(.data, …)
Compute table of summaries. Also **summarise_**().
summarise(mtcars, avg = mean(mpg))

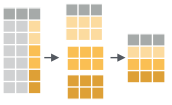
count(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in … Also **tally**().
count(iris, Species)

VARIATIONS

summarise_all() - Apply funs to every column.
summarise_at() - Apply funs to specific columns.
summarise_if() - Apply funs to all cols of one type.

Group Cases

Use **group_by()** to created a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.



```
mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))
```

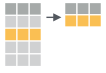
group_by(.data, ..., add = FALSE)
Returns copy of table grouped by …
g_iris <- group_by(iris, Species)

ungroup(x, …)
Returns ungrouped copy of table.
ungroup(g_iris)

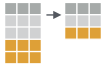
Manipulate Cases

EXTRACT CASES

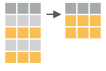
Row functions return a subset of rows as a new table. Use a variant that ends in _ for non-standard evaluation friendly code.



filter(.data, …) Extract rows that meet logical criteria. Also **filter_()**. *filter(iris, Sepal.Length > 7)*

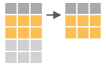


distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values. Also **distinct_()**.
distinct(iris, Species)



sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select fraction of rows.
sample_frac(iris, 0.5, replace = TRUE)

sample_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select size rows. *sample_n(iris, 10, replace = TRUE)*



slice(.data, …) Select rows by position. Also **slice_()**. *slice(iris, 10:15)*

top_n(x, n, wt) Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

Logical and boolean operators to use with filter()

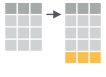
<	<=	is.na()	%in%	|	xor()
>	>=	!is.na()	!	&	

See **?base::logic** and **?Comparison** for help.

ARRANGE CASES

arrange(.data, …)
Order rows by values of a column (low to high), use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

ADD CASES



add_row(.data, ..., .before = NULL, .after = NULL)
Add one or more rows to a table.
add_row(faithful, eruptions = 1, waiting = 1)

Column functions return a set of columns as a new table. Use a variant that ends in _ for non-standard evaluation friendly code.

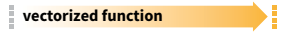
select(.data, …)
Extract columns by name. Also **select_if()**
select(iris, Sepal.Length, Species)

Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

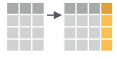
contains(match)	**num_range**(prefix, range)	**:**, e.g. mpg:cyl
ends_with(match)	**one_of**(…)	**-**, e.g, -Species
matches(match)	**starts_with**(match)	

MAKE NEW VARIABLES

These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).



vectorized function



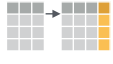
mutate(.data, …)
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

transmute(.data, …)
Compute new column(s), drop others.
transmute(mtcars, gpm = 1/mpg)

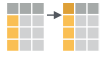
mutate_all(.tbl, .funs, …) Apply funs to every column. Use with **funs()**.
mutate_all(faithful, funs(log(.), log2(.)))

mutate_at(.tbl, .cols, .funs, …) Apply funs to specific columns. Use with **funs()**, **vars()** and the helper functions for select().
mutate_at(iris, vars(-Species), funs(log(.)))

mutate_if(.tbl, .predicate, .funs, …)
Apply funs to all columns of one type. Use with **funs()**.
mutate_if(iris, is.numeric, funs(log(.)))



add_column(.data, ..., .before = NULL, .after = NULL) Add new column(s).
add_column(mtcars, new = 1:32)



rename(.data, …) Rename columns.
rename(iris, Length = Sepal.Length)

RStudio Cheatsheets

Vectorized Functions

TO USE WITH MUTATE ()

mutate() and **transmute()** apply vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

vectorized function

OFFSETS

dplyr::**lag()** - Offset elements by 1
dplyr::**lead()** - Offset elements by -1

CUMULATIVE AGGREGATES

dplyr::**cumall()** - Cumulative all()
dplyr::**cumany()** - Cumulative any()
cummax() - Cumulative max()
dplyr::**cummean()** - Cumulative mean()
cummin() - Cumulative min()
cumprod() - Cumulative prod()
cumsum() - Cumulative sum()

RANKINGS

dplyr::**cume_dist()** - Proportion of all values <=
dplyr::**dense_rank()** - rank with ties = min, no gaps
dplyr::**min_rank()** - rank with ties = min
dplyr::**ntile()** - bins into n bins
dplyr::**percent_rank()** - min_rank scaled to [0,1]
dplyr::**row_number()** - rank with ties = "first"

MATH

+, - , *, /, ^, %/%, %% - arithmetic ops
log(), log2(), log10() - logs
<, <=, >, >=, !=, == - logical comparisons

MISC

dplyr::**between()** - x >= left & x <= right
dplyr::**case_when()** - multi-case if_else()
dplyr::**coalesce()** - first non-NA values by element across a set of vectors
dplyr::**if_else()** - element-wise if() + else()
dplyr::**na_if()** - replace specific values with NA
pmax() - element-wise max()
pmin() - element-wise min()
dplyr::**recode()** - Vectorized switch()
dplyr::**recode_factor()** - Vectorized switch() for factors

Summary Functions

TO USE WITH SUMMARISE ()

summarise() applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

summary function

COUNTS

dplyr::**n()** - number of values/rows
dplyr::**n_distinct()** - # of uniques
sum(!is.na()) - # of non-NA's

LOCATION

mean() - mean, also **mean(!is.na())**
median() - median

LOGICALS

mean() - Proportion of TRUE's
sum() - # of TRUE's

POSITION/ORDER

dplyr::**first()** - first value
dplyr::last() - last value
dplyr::nth() - value in nth location of vector

RANK

quantile() - nth quantile
min() - minimum value
max() - maximum value

SPREAD

IQR() - Inter-Quartile Range
mad() - mean absolute deviation
sd() - standard deviation
var() - variance

Row Names

Tidy data does not use rownames, which store a variable outside of the columns. To work with the rownames, first move them into a column.

rownames_to_column()
Move row names into col.
a <- rownames_to_column(iris, var = "C")

column_to_rownames()
Move col in row names.
column_to_rownames(a, var = "C")

Also has_**rownames()**, **remove_rownames()**

Combine Tables

COMBINE VARIABLES

Use **bind_cols()** to paste tables beside each other as they are.

bind_cols(…) Returns tables placed side by side as a single table.
BE SURE THAT ROWS ALIGN.

Use a "**Mutating Join**" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…)
Join matching values from y to x.

right_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…)
Join matching values from x to y.

inner_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…)
Join data. Retain only rows with matches.

full_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…)
Join data. Retain all values, all rows.

Use **by = c("col1", "col2")** to specify the column(s) to match on.
left_join(x, y, by = "A")

Use a named vector, **by = c("col1" = "col2")**, to match on columns with different names in each data set.
left_join(x, y, by = c("C" = "D"))

Use **suffix** to specify suffix to give to duplicate column names.
left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))

COMBINE CASES

Use **bind_rows()** to paste tables below each other as they are.

bind_rows(…, .id = NULL)
Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured)

intersect(x, y, …)
Rows that appear in both x and z.

setdiff(x, y, …)
Rows that appear in x but not z.

union(x, y, …)
Rows that appear in x or z. (Duplicates removed). union_all() retains duplicates.

Use **setequal()** to test whether two data sets contain the exact same rows (in any order).

EXTRACT ROWS

Use a "**Filtering Join**" to filter one table against the rows of another.

semi_join(x, y, by = NULL, …)
Return rows of x that have a match in y. USEFUL TO SEE WHAT WILL BE JOINED.

anti_join(x, y, by = NULL, …)
Return rows of x that do not have a match in y. USEFUL TO SEE WHAT WILL NOT BE JOINED.

RStudio